



The Azure Cloud-Native Architecture Mapbook

Explore Microsoft's cloud infrastructure, application, data, and security architecture

Stéphane Eyskens | Ed Price



The Azure Cloud-Native Architecture Mapbook

Explore Microsoft's cloud infrastructure, application, data, and security architecture

Stéphane Eyskens

Ed Price

Packt

BIRMINGHAM—MUMBAI

The Azure Cloud-Native Architecture Mapbook

Copyright © 2021 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author(s), nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Aaron Lazar

Publishing Product Manager: Denim Pinto

Senior Editor: Storm Mann

Content Development Editor: Nithya Sadanandan

Technical Editor: Gaurav Gala

Copy Editor: Safis Editing

Project Coordinator: Deeksha Thakkar

Proofreader: Safis Editing

Indexer: Rekha Nair

Production Designer: Nilesh Mohite

First published: February 2021

Production reference: 1160221

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80056-232-5

www.packt.com

To my wife, Diana Lucic, who designed all the maps of this book.

– Stéphane Eyskens

*To my sons, Ben and Yoav, for showing me how talent and creativity evolve.
To Tsippi and Shlomo Bobbe, for their love, support, and inspiration.*

– Ed Price

Contributors

About the authors

Stéphane Eyskens is an Azure solutions architect and a digital transformation advocate, helping organizations to get better results out of their cloud investments. As an MVP, he is an active contributor to the Microsoft Tech Community and has worked on multiple open source projects available on GitHub. Stéphane is also a Pluralsight assessment author, as well as the author of multiple books and online recordings.

Ed Price is a senior program manager in engineering at Microsoft, with an MBA in technology management. He leads Microsoft's efforts to publish reference architectures on the Azure Architecture Center (<http://aka.ms/Architectures>). Previously, he drove data center deployment and customer feedback, and he ran Microsoft's customer feedback programs for Azure development, Service Fabric, IoT, Functions, and Visual Studio. He was also a technical writer at Microsoft for 6 years and helped lead TechNet Wiki. He is the co-author of five books, including *Learn to Program with Small Basic* and *ASP.NET Core 5 for Beginners*, available from Packt.

About the reviewers

Giorgos-Chrysovalantis Grammatikos is an Azure solutions architect with Tisiski Ltd. He is an IT pro with over 10 years of experience in the industry, has achieved various Azure and other Microsoft certifications, and has been an Azure MVP since 2018. His specialization is in various Microsoft technologies including the Azure cloud, SQL Server, Power BI, and Hyper-V. He is an active member of the Microsoft community, blogging on his website cloudopszone.com and posting technical articles on the Microsoft wiki and dev blogs. He is a frequent public speaker at meetups and various Microsoft Azure events.

Sjoukje Zaal is a Microsoft chief technical officer at Capgemini, a Microsoft regional director, and a Microsoft Azure MVP with over 20 years of experience providing architecture, development, consultancy, and design expertise. She mainly focuses on cloud, security, productivity, and IoT.

She loves to share her knowledge and is active in the Microsoft community as a co-founder of the user groups Tech Daily Chronicle, Global XR Community, and the Mixed Reality User Group. She is also a board member of Azure Thursdays and Global Azure. Sjoukje is an international speaker, is involved in organizing many events, and has written several books and blogs. Sjoukje is also part of the MVP Diversity and Inclusion Advisory Board.



```
function signUp( service ) {  
  if ( service.type === 'Azure' &&  
      service.accountsFree === true ){  
    return true;  
  }  
}  
signUp( Azure ); // true
```

Let's go.

Make your vision real.
Start experimenting with
free cloud services.
[Start free >](#)

Get help with
your project.
[Talk to a
sales specialist >](#)

Table of Contents

Preface			vii
Section 1: Solution and Infrastructure			1
1			
Getting Started as an Azure Architect			3
Technical requirements	4	FaaS (Function as a Service)	14
Getting to know architectural duties	4	CaaS (Containers as a Service)	15
Enterprise architects	4	DBaaS (Database as a Service)	16
Domain architects	5	XaaS or *aaS (Anything as a Service)	16
Solution architects	5	Introducing Azure architecture maps	17
Data architects	6	How to read a map	18
Technical architects	7	Understanding the key factors of a successful cloud journey	19
Security architects	8	Defining the vision with the right stakeholders	19
Infrastructure architects	10	Defining the strategy with the right stakeholders	20
Application architects	10	Starting implementation with the right stakeholders	20
Azure architects	11	Practical scenario	21
Architects versus engineers	11	Summary	25
Getting started with the essential cloud vocabulary	12		
Cloud service models map	12		
IaaS (Infrastructure as a Service)	13		
PaaS (Platform as a Service)	13		

2

Solution Architecture 27

Technical requirements	28	Solution architecture use case	55
The solution architecture map	28	Looking at a business scenario	55
Zooming in on the different workload types	30	Using keywords	55
Understanding systems of engagement	30	Using the solution architecture map against the requirements	56
Understanding systems of record	32	Building the target reference architecture	58
Understanding systems of insight	34	Code view of our workflow-based reference architecture	62
Understanding systems of interaction (IPaaS)	36	Looking at the code in action	67
Looking at cross-cutting concerns and non-functional requirements	41	Understanding the gaps in our reference architecture	72
Looking at cross-cutting concerns and the cloud journey	52	Summary	73
Zooming in on containerization	52		

3

Infrastructure Design 75

Technical requirements	76	AKS infrastructure	97
The Azure infrastructure architecture map	76	Exploring networking options with AKS	99
Zooming in on networking	78	Exploring deployment options with AKS	104
The most common architecture	80	Monitoring AKS	106
Data center connectivity options	81	Exploring AKS storage options	106
Zoning	82	Scaling AKS	107
Routing and firewalling	83	Exploring miscellaneous aspects	108
Zooming in on monitoring	84	AKS and service meshes for microservices versus Azure native services	109
Zooming in on high availability and disaster recovery	90	AKS reference architecture for microservices – cluster boundaries	112
Zooming in on backup and restore	94	AKS reference architecture for microservices – cluster internals	116
Zooming in on HPC	96	Summary	118

4

Infrastructure Deployment 119

Technical requirements	120	Understanding the ARM template deployment methods	137
Introducing Continuous Integration and Continuous Deployment (CI/CD)	120	Understanding the ARM template deployment scopes	138
Introducing the CI/CD process	121	Understanding the ARM template deployment modes	142
Introducing the IaC CI/CD process	122	Understanding the anatomy of an ARM template	144
The Azure deployment map	124	Building a concrete example using linked templates	147
Getting started with the Azure CLI, PowerShell, and Azure Cloud Shell	127	Getting started with Azure Bicep	159
Playing with the Azure CLI from within Azure Cloud Shell	127	Getting started with Terraform	162
Using PowerShell from within Azure Cloud Shell	132	Zooming in on a reference architecture with Azure DevOps	168
Combining PowerShell and the Azure CLI from within Azure Cloud Shell	134	Using a simple approach to an IaC factory	169
Understanding the one that rules them all	135	Using an advanced approach to an IaC factory	172
Diving into ARM templates	137	Summary	175
Getting started with ARM	137		

Section 2: Application Development, Data, and Security 177

5

Application Architecture 179

Technical requirements	180	Zooming in on data	185
Understanding cloud and cloud-native development	181	Zooming in on cloud design patterns	186
Exploring the Azure Application Architecture Map	183	Dealing with cloud-native patterns	193
		Understanding the COMMODITIES top-level group	201

Exploring EDAs	204	Understanding Dapr components	219
Inspecting the Azure Service Bus configuration	210	Getting started with Dapr SDKs	220
Adding the other components to the mix	214	Looking at our scenario	222
Developing microservices	216	Developing our solution	223
Using Dapr for microservices	217	Testing our solution	229
		Combining Dapr and the API gateway of Azure APIM	232
		Summary	238

6

Data Architecture **239**

Technical requirements	240	Introducing AI solutions	253
Looking at the data architecture map	240	Understanding machine learning and deep learning	254
Analyzing traditional data practices	242	Integrating AI solutions	256
Introducing the OLAP and OLTP practices	243	Dealing with other data concerns	257
Introducing the ETL practice	243	Introducing Azure Cognitive Search	257
Introducing the RDBMS practice	244	Sharing data with partners and customers (B2B)	258
Delving into modern data services and practices	245	Migrating data	258
Introducing the ELT practice	246	Governing data	259
Exploring NoSQL services	246	Getting our hands dirty with a near real-time data streaming use case	259
Learning about object stores	248	Setting up the Power BI workspace	260
Diving into big data services	249	Setting up the Azure Event Hubs instance	260
Ingesting big data	250	Setting up Stream Analytics (SA)	261
Exploring big data analytics	251	Testing the code	263
Azure-integrated open source big data solutions	253	Summary	266

7

Security Architecture 267

Technical requirements	268	Delving into the most recurrent Azure security topics	294
Introducing cloud-native security	268	Exploring Azure managed identities in depth	294
Reviewing the security architecture map	270	Demystifying SAS	297
Exploring the recurrent services security features	272	Understanding APL and its impact on network flows	298
Exploring the recurrent data services security features	280	Understanding Azure resource firewalls	301
Zooming in on encryption	282	Adding the security bits to our Contoso use case	302
Managing your security posture	286	Summary	308
Zooming in on identity	290		

Section 3: Summary 309

8

Summary and Industry Scenarios 311

Revisiting our architectures	312	Banking and financial services scenarios	323
Sample architecture	312	Banking system cloud transformation	323
Solution architecture	313	Decentralized trust using blockchain	324
Infrastructure architecture	315	Additional financial services architectures	324
Azure deployment	316	Gaming scenarios	325
Application architecture	317	Low-latency multiplayer gaming	326
Data architecture	318	Gaming using MySQL or Cosmos DB	326
Security architecture	320	Healthcare scenarios	326
Visiting the verticals	321	Building a telehealth system on Azure	327
Automotive and transportation scenarios	321	Medical data storage architectures	327
Predictive insights with vehicle telematics	321	AI healthcare solutions	328
Predictive aircraft engine monitoring	322	Predicting length of stay using SQL Server R Services	328
IoT analytics for autonomous driving	323		

Producing and consuming IoT healthcare data	328	Retail scenarios	333
Confidential computing on a healthcare platform	329	Retail and e-commerce Azure database architectures	333
Manufacturing scenarios	329	Demand forecasting with Spark on HDInsight	334
Supply chain track and trace	330	Demand forecasting with machine learning	334
Industrial IoT analytics	330	AI retail scenarios	334
AI and analytics manufacturing architectures	330	Architecture for buy online, pick up in store	335
Oil and gas scenarios	331	The unique values of this book	336
Run reservoir simulation software on Azure	331	Summary	337
Oil and gas tank level forecasting	332	Why subscribe?	339
IoT monitor and manage loops	332		
Other Books You May Enjoy			340
Index			343

Preface

Have you ever visited a large city on your own? Sometimes you get lost, and sometimes you lose time. However, you can make your trip more valuable by taking an expert-guided tour. That's what this book is: an expert-guided tour of Azure. Our different maps will be your compass to sail the broad Azure landscape. The Microsoft cloud platform offers a wide range of services, providing a million ways to architect your solutions. This book uses original maps and expert analysis to help you explore Azure and choose the best solutions for your unique requirements. Beyond maps, the book is inspired by real-world situations and challenges. We will share typical and cross-industry concerns to help you become a better Azure architect. Our real-world-inspired architecture diagrams and use cases should put you in a better position to tackle your own challenges. Although an architecture role may be high-level, we also wanted to dive deeper into some topics and make you work harder on some use cases. In this respect, you will have some hands-on work to do too. However, our primary objective is to make you stronger in the various architecture disciplines that are scoped to Azure and that every Azure architect should be comfortable with.

Who this book is for

This book is intended for aspiring and confirmed Azure architects. This book is broad and encompasses multiple architecture disciplines and concepts, so you should ideally have a broad skillset to enjoy the book. Nevertheless, IT engineers and developers will also ramp up their knowledge and find value in this book.

What this book covers

Chapter 1, Getting Started as an Azure Architect, starts by sharing a view of the different architecture disciplines. We define the roles and responsibilities of the various architects (enterprise, solution, infrastructure, data, and security). The rationale of going through these definitions lies in the fact that, from our experience, we have noticed some knowledge gaps in what the different stakeholders are doing. This often leads to turf wars, which can be avoided simply by understanding the broader picture. We then introduce our maps, and we help you understand how to properly conduct a cloud strategy and what the key aspects are that will make your cloud journey successful. In a nutshell, we give you a glimpse into what it feels like to be an Azure architect who has to deal with all these different disciplines, and who sometimes must report to top management on strategic aspects.

Chapter 2, Solution Architecture, covers key aspects to consider when building a cloud solution. A solution architect is responsible for the end-to-end aspects of a solution, from its development to its monitoring. A solution architect knows what Agile methodologies are, as well as what ITIL, TOGAF, and COBIT are. They are the cornerstone of a solution, its main pillar. The primary role of a solution architect is to assemble all the building blocks to make a consistent and coherent design, as well as to talk to various stakeholders. Their stakeholders are other, more specialized architects, developers, and IT engineers, as well as enterprise architects and management. This chapter remains high-level from a technical perspective because we will still envision Azure as a whole. We share the solution architecture map, which encompasses many Azure services, and we explore multiple dimensions around the non-functional requirements. We also zoom in on Azure's container platform offering, which has been booming and expanding greatly over the last few years. Lastly, we will walk you through a concrete use case and a glimpse into what comes next, including a deeper dive into the technical and technological aspects.

Chapter 3, Infrastructure Design, delves deeper into technical matters. We will review the typical infrastructure topologies and we will zoom into infrastructure-specific concerns such as networking, monitoring, backup and restore, high availability, and disaster recovery (for which we'll see a sample use case). Because containerization has become mainstream, we will also dive into **Azure Kubernetes Services (AKS)** and unveil a dedicated AKS architecture map. You will learn that AKS is not really a service like the others, and we will walk you through a reference architecture to host a service mesh (for microservices) in AKS.

Chapter 4, Infrastructure Deployment, is almost entirely hands-on! You will learn about the different **Infrastructure as Code (IaC)** tools and frameworks. You will provision some Azure services using Azure Resource Manager templates, Bicep, and Terraform. Nevertheless, we won't forget our architecture glasses, so we will also look at the machinery of a **Continuous Integration and Continuous Delivery/Deployment (CI/CD)** factory.

Chapter 5, Application Architecture, looks at what the development architecture would look like for building an app on the Microsoft cloud. You may ask 10 different people what cloud-native means, and you might receive 10 different answers. So, we will start by explaining what we mean when we refer to the cloud and cloud-native solutions. Next, we will review some modern design patterns, such as CQRS, Event Sourcing, and so on. In the process, we will map them to the Azure services to help you identify how to bundle the services together in order to build solutions based on these patterns. Lastly, we will go through a microservices use case, using **Dapr (Distributed Application Runtime)**, which is a very recent and promising framework for developing distributed applications. Throughout this chapter, our motto will be to *not reinvent the wheel*. Instead, leverage the ecosystem to design and build your solutions.

Chapter 6, Data Architecture, explores how data is processed and stored. Data is the new gold, and Azure contains many gold mines! In this chapter, we will consider traditional and modern data practices in opposition to each other, and see how to use both in Azure. We will also explore big data and artificial intelligence and analytics. At last, our hands-on use case is based on a data-streaming scenario. We are going to build a real-time dashboard, which consolidates aggregates of metrics from a fake speed detector (which we have developed for you). A separate real-time tile will show all the vehicles that should receive a fine (for breaking the law).

Chapter 7, Security Architecture, emphasizes and explains the importance of security in the cloud. Security is everywhere, and it's even more important with the cloud. This tends to awaken age-old fears and trepidations. This topic certainly deserves an entire book, so (to avoid writing a second book) we decided to be very pragmatic and to focus on the essential parts only. We start by giving you a glimpse into cloud-native security, to see beyond the technology and what the required mindset is. We will then explain why there is a paradigm shift in identity with the public cloud, by simply ... proving it! Lastly, we will focus on the most recurrent security services and topics in Azure, which you must absolutely master as an Azure architect. Throughout the chapter, our motto will be to *not simply stack network layers*. Instead, think further and modernize your security practices.

Chapter 8, Summary and Industry Scenarios, revisits the topics covered in the book and consolidates our key ideas from each previous chapter. In other words, we'll identify what the most important aspects to remember are. In addition, we'll look at several key industry verticals through the lens of the previous chapters, to guide you through some existing architectures that you can continue exploring after you complete the book. We'll finish with some notes on the unique key values of this book, and a brief summary.

To get the most out of this book

To enjoy the book and practice our hands-on exercises, you will of course need an Azure subscription, and for the bravest readers (those who are implementing our use cases), you'll also need Docker, Visual Studio 2019, and/or Visual Studio Code. All our code samples are built in .NET Core. From a higher-level perspective, you'll be able to quickly grasp the concepts in this book if you're already an architect or a senior developer/IT pro. Don't worry if you need to use Google from time to time (to look up names and terms); it's perfectly normal, as we explore the main architectural dimensions. (We do explain all the basic concepts, but to keep the content focused for senior developers, senior IT pros, and new architects, the book is written with a certain expectation of technical knowledge.)

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (the link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the maps, diagrams, and sample code for this book from GitHub at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Code in Action

Code in Action videos for this book can be viewed at <http://bit.ly/3pp9vIH>.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781800562325_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Locate and open the `appsettings.json` file, in the `netcoreapp3.1` folder."

A block of code is set as follows:

```
public class DataObject{
    private string[] sensorNames = new string[] { "Brussels",
        "Genval" };
    public string sensorName { get; private set; }
    public double speed { get; private set; }
    public string plateNumber { get; private set; }
    public DataObject()
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in **bold**:

```
public class DataObject{
    private string[] sensorNames = new string[] {
        "Brussels", "Genval" };
    public string sensorName { get; private set; }
    public double speed { get; private set; }
    public string plateNumber { get; private set; }
    public DataObject()
```

Any command-line input or output is written as follows:

```
$ az storage account list
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Choose the **Custom Streaming** data tile type."

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customer-care@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, select your book, click the Errata Submission Form link, and enter the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

Section 1: Solution and Infrastructure

In this section, we will first revise the different architecture practices and dimensions, because the word architect is sometimes misunderstood or abused. We will share our views on the different architecture dimensions that we cover throughout the book. The initial chapter will set the scene for your journey. Then, we will get you started with Azure and acquainted with the services that you can assemble to design and build solutions. We will also focus on an essential foundation – the infrastructure – and you will get to know how to leverage Infrastructure as Code and automation, to get an optimal return on investment out of your cloud expenditures.

In this section, we will cover the following topics:

- *Chapter 1, Getting Started as an Azure Architect*
- *Chapter 2, Solution Architecture*
- *Chapter 3, Infrastructure Design*
- *Chapter 4, Infrastructure Deployment*

1

Getting Started as an Azure Architect

In this chapter, we will focus on what an architect's role entails and explain the various cloud service models that are made available by the Microsoft Azure platform. We will describe how the numerous maps in this book are built, what they intend to demonstrate, and how to make sense of them.

More specifically, in this chapter, we will cover the following topics:

- Getting to know architectural duties
- Getting started with the essential cloud vocabulary
- Introducing Azure architecture maps
- Understanding the key factors of a successful cloud journey

Our purpose is to help you learn the required vocabulary that is used across the book. You will also understand the duties of an Azure architect. We will explain the most frequently used service models and their typical associated use cases, which every Azure architect should know. We start smoothly, but beware that the level of complexity will increase as we go. Let's start by getting acquainted with the definition of an architect.

Technical requirements

The Maps provided in this chapter are available at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/tree/master/Chapter01>.

Getting to know architectural duties

Before we define what an Azure architect is, let's first define what an architect's *role* is and how our maps specialize to reflect these different profiles. The word **architect** is used everywhere on the IT planet. Many organizations have their own expectations when it comes to defining the tasks and duties of an architect. Let's share our own definitions as well as some illustrative diagrams.

Enterprise architects

Enterprise architects oversee the IT and business strategies, and they make sure that every IT initiative is in line with the enterprise business goals. They are directly reporting to the IT leadership and are sometimes scattered across business lines. They are also the guardians of building coherent and consistent overall IT landscapes for their respective companies. Given their broad role, enterprise architects have a helicopter view of the IT landscape, and they are not directly dealing with deep-dive technical topics, nor are they looking in detail at specific solutions or platforms, such as Azure, unless a company would put all its assets in Azure. In terms of modeling, they often rely on the **TOGAF** (short for **The Open Group Architecture Framework**) modeling framework and ArchiMate. The typical type of diagrams they deal with looks like the following:

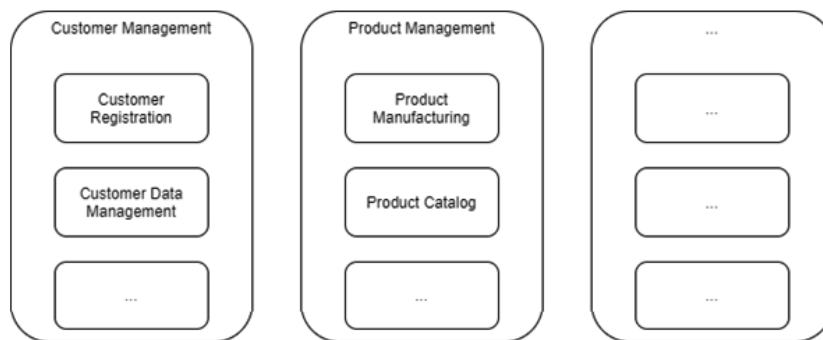


Figure 1.1 – Capability viewpoint: ArchiMate

As you can see, this is very high level and not directly related to any technology or platform. Therefore, this book focusing on Azure is not intended for enterprise architects, but they are, of course, still welcome to read it!

Domain architects

Domain architects own a single domain, such as the cloud. In this case, the cloud is broader than just Azure, as it would probably encompass both public and private cloud providers. Domain architects are tech-savvy, and they define their domain roadmaps while supervising domain-related initiatives. Compared to enterprise architects, their scope is more limited, but it is still too broad to master the bits and bytes of an entire domain. This book, and more particularly our generic maps, will certainly be of great interest for cloud domain architects. Diagram-wise, the domain architects will also rely on TOGAF and other architecture frameworks, but scoped to their domain.

Solution architects

Solution architects help different teams to build solutions. They have *T-shaped* skills, which means that they are specialists in a given field (the base of the *T*), but they can also collaborate across disciplines with the other experts (the top of the *T*). Solution architects are usually in charge of designing solution diagrams, and they tackle non-functional requirements, such as security, performance, and scalability. Their preferred readings will be our chapter dedicated to solution architecture, as well as some reference architectures. Solution architects may build both high-level technology-agnostic, and platform-specific diagrams. Azure solution architects may build reference architectures, such as, for instance, one that we can find on the Azure Architecture Center (<https://docs.microsoft.com/azure/architecture/>):

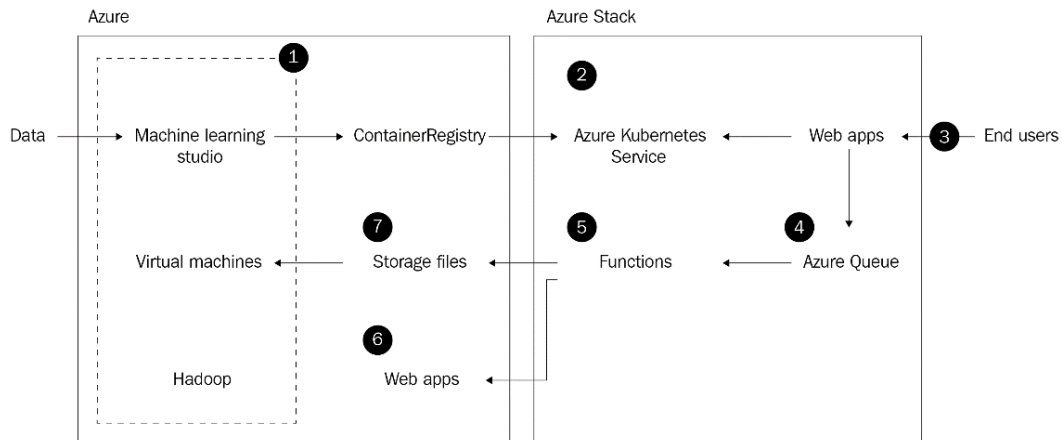


Figure 1.2 – AI at the edge with Azure Stack

The preceding diagram illustrates how to leverage both Azure Stack and Azure, together with artificial intelligence services. Such architectures can be instantiated per asset, but still remain rather high-level. They depict the components and their interactions, and must be completed by the non-functional requirements. We will explore this in more detail in the next chapter.

Data architects

Data architects oversee the entire data landscape. They mostly focus on designing data platforms, for storage, insights, and advanced analytics. They deal with data modeling, data quality, and business intelligence, which consists of extracting valuable insights from the data, in order to realize substantial business benefits. A well-organized data architecture should ultimately deliver the **DIKW (Data, Information, Knowledge, Wisdom)** pyramid as shown in *Figure 1.3*:

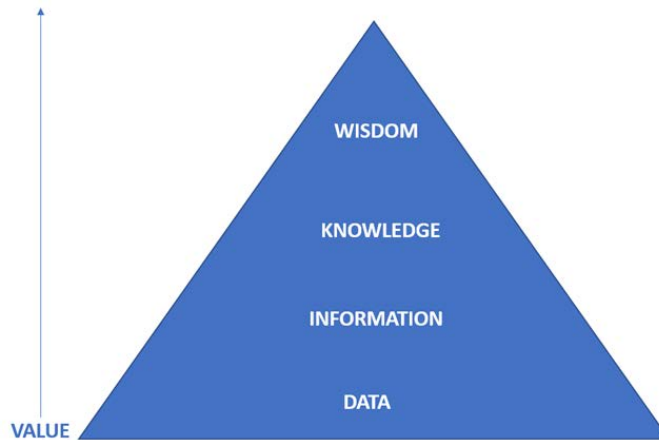


Figure 1.3 – DIKW pyramid

Organizations have a lot of data, from which they try to extract valuable information, knowledge, and wisdom over time. The more you climb the pyramid, the higher the value. Consider the following scenario to understand the DIKW pyramid:

DATA	31, 3, 3,000
INFORMATION	DAY: 31, MONTH: MARCH, VISITS: 3000
KNOWLEDGE	Up to 3000 users visited our website on March 31, which is way above the usual daily average of 650. March 31 seems always busy, year after year.
WISDOM	We will make sure to restock warehouses before March 31.

Figure 1.4 – DIKW pyramid example

Figure 1.4 shows that we start with raw data, which does not really make sense without context. These are just numbers. At the information stage, we understand that 31 stands for a day, 3 is March, and 3,000 is the number of visits. Now, these numbers mean something. The knowledge block is self-explanatory. We have analyzed our data and noticed that year after year, March 31 is a busy day. Thanks to this valuable insight, we can take the wise decision to restock our warehouses up front to make sure we do not run short on goods.

That is, among other things, the work of a data architect to help organizations learn from their data.

Data is the new gold rush, and Azure has a ton of data services as part of its catalog, which we will cover in *Chapter 6, Data Architecture*.

Technical architects

Technical architects have a deep vertical knowledge of a platform or technology stack, and they have hands-on practical experience. They usually coach developers, IT professionals, and DevOps engineers in the day-to-day implementation of a solution. They contribute to reference architectures by zooming inside some of the high-level components. For instance, if a reference architecture, designed by a solution architect, uses **Azure Kubernetes Services (AKS)** as part of the diagram, the technical architect might zoom inside the AKS cluster, to bring extra information on the cluster internals and some specific technologies. To illustrate this, *Figure 1.5* shows a high-level diagram a solution architect might have done:

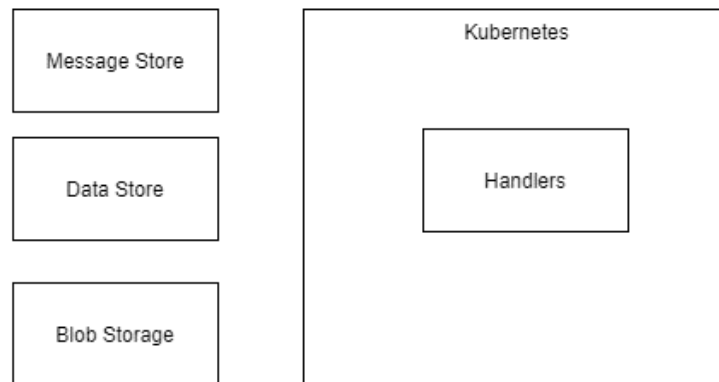


Figure 1.5 – Reference architecture example

Figure 1.6 shows the extra contribution of a technical architect:

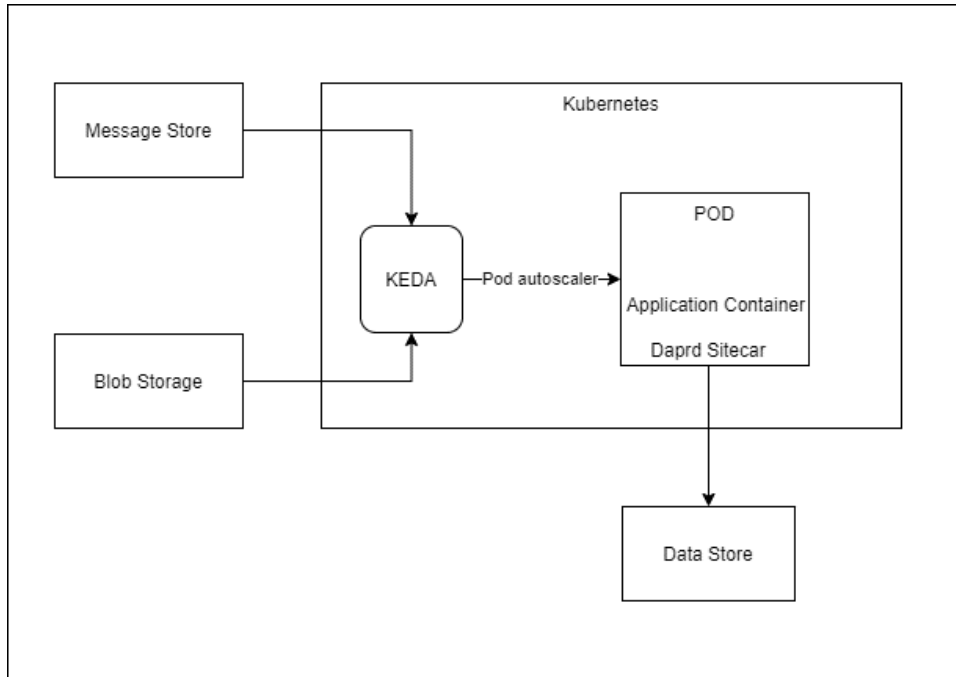


Figure 1.6 – Reference architecture refined by a technical architect

We see that the reviewed diagram contains precise technologies, such as **KEDA (Kubernetes-based Event Driven Autoscaling)**, and **Dapr (Distributed Application Runtime)**, for both autoscaling and interactions with event and data stores.

The technical architect will mostly be interested in our detailed maps.

Security architects

In this hyper-connected world, the importance of security architecture has grown a lot. **Security architects** have a vertical knowledge of the security field. They usually deal with regulatory or in-house compliance requirements. The cloud and, more particularly, the public cloud, often emphasizes security concerns (much more than for equivalent on-premises systems and applications). With regard to diagrams, security architects will add a security view (or request one) to the reference solution architectures, such as the following:

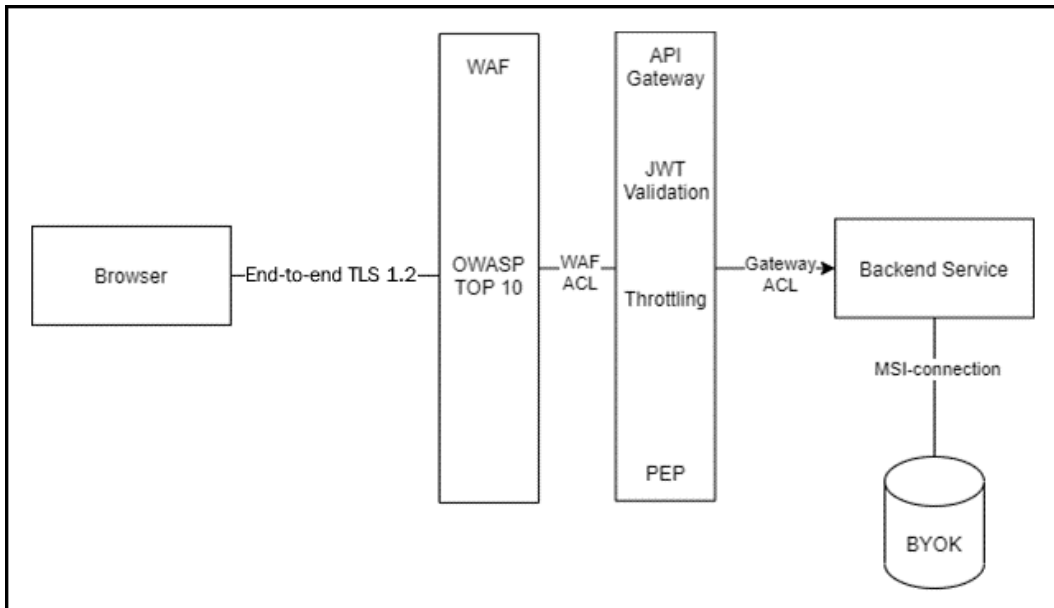


Figure 1.7 – Simplified security view example

In the preceding example, the focus is set on pure security concerns: encryption in transit (TLS 1.2) between the browser and the **web application firewall (WAF)**. The WAF enforces a security ruleset to protect against the **Open Web Application Security Project (OWASP)** top 10 vulnerabilities. The API gateway acts as a policy enforcement point before it forwards the request to the backend service. The backend authenticates to the database using **managed service identity (MSI)**, and the database is encrypted at REST with customer-managed keys. *Figure 1.7* clearly emphasizes what security architects are interested in.

However, as we will explore further in *Chapter 7, Security Architecture*, mastering cloud and cloud-native security is a tough challenge for a traditional (on-premises) security architect. Cloud native's defense in depth primarily relies on identity, while traditional defense in depth relies heavily on the network perimeter. This gap is often a source of tension between the cloud and non-cloud worlds.

Infrastructure architects

Infrastructure architects focus on building IT systems that host applications, or systems that are sometimes shared across workloads. They play a prominent role in setting up hybrid infrastructures, which bridge both the cloud and the on-premises world. Their diagrams reflect an infrastructure-only view, often related to the concept of a landing zone, which consists of defining how and where business assets will be hosted. A typical infrastructure diagram that comes to mind for a hybrid setup is the **Hub & Spoke** architecture (<https://docs.microsoft.com/azure/architecture/reference-architectures/hybrid-networking/hub-spoke>):

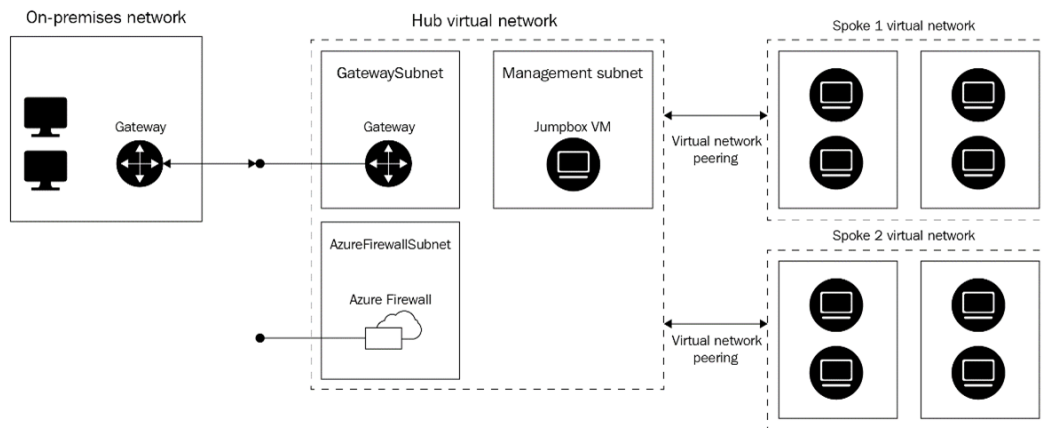


Figure 1.8 – Hub and spoke architecture

Figure 1.8 is a simplified view of the hub and spoke, which, in reality, is much more complex than this. We will explore this further in *Chapter 3, Infrastructure Design*. We will also stress some important aspects related to legacy processes, so as to maximize the chances of a successful cloud journey.

Application architects

Application architects focus on building features that are requested by the business. Unlike other architects, they are not primarily concerned by non-functional requirements. Their role is to enforce industry best practices and coding patterns in order to make maintainable and readable applications. Their primary concerns are to integrate with the various Azure services and SDKs, as well as leverage cloud and cloud-native design patterns that are immensely different from traditional systems. Beyond this book, a good source of information for them is the Microsoft documentation on cloud design patterns (<https://docs.microsoft.com/azure/architecture/patterns/>).

What is challenging for application architects is to correctly understand the ecosystem in which the application runs. Today, there is a clear trend that entails breaking the monolith. In other words, we slice the architecture into multiple decoupled pieces, and we end up with a very distributed architecture. In most modern applications, a lot of common duties are offloaded to specialized services, often not so well known by old school application architects. For instance, an API gateway already has built-in policies for API throttling, token validation, and caching. Instead of writing your own plumbing in code to handle this, it is better to offload it. Another attention point for application architects is the horizontal scaling story of the cloud, meaning that applications/services must be multi-instance aware, which is rarely the case with monoliths. We will explore these concerns further in *Chapter 5, Application Architecture*.

Azure architects

From the top to the bottom of our enumeration, the IT landscape shrinks, from the broadest to the narrowest scope. It would be very surprising to ever meet an Azure *enterprise* architect. Similarly, it is unlikely that we will stumble upon an Azure *domain* architect, since the parent domain would rather be the cloud (which is much broader than just Azure).

However, it makes sense to have **Azure-focused solution architects**, technical architects, and data architects, because they get closer to the actual implementation of a solution or platform. Depending on your interest and background, you might specialize in one or more service models, which are depicted in the following section. Thus, some Azure architects will be interested in specialized maps, and some simply won't be interested, although it is always highly recommended to look at the broader picture.

Architects versus engineers

Before we move on, we need to address the engineer that we all have inside of us! What differentiates architects from engineers is probably the fact that most architects have to deal with the non-functional requirements piece. In contrast, engineers, such as developers and IT professionals, are focused on delivering and maintaining the features and systems requested by the business, which makes them very close to the final solution. Nevertheless, this book also contains some sections that are likely to help engineers build effective solutions.

Now that we are clear with what the role of an architect is all about, it is time to get started with the different service models and acquire the essential vocabulary that every Azure architect should know.

Getting started with the essential cloud vocabulary

In this section, we will cover the essential basic skills every Azure architect should have. The cloud has different service models, which all serve different purposes. It is very important to understand the advantages and inconveniences of each model, and to get acquainted with the jargon relating to the cloud.

Cloud service models map

Figure 1.9 demonstrates our first map (not counting our sample map), which depicts the different cloud service models and introduces some vocabulary. This map features two additional dimensions (costs and ops) to each service model, as well as some typical use cases:

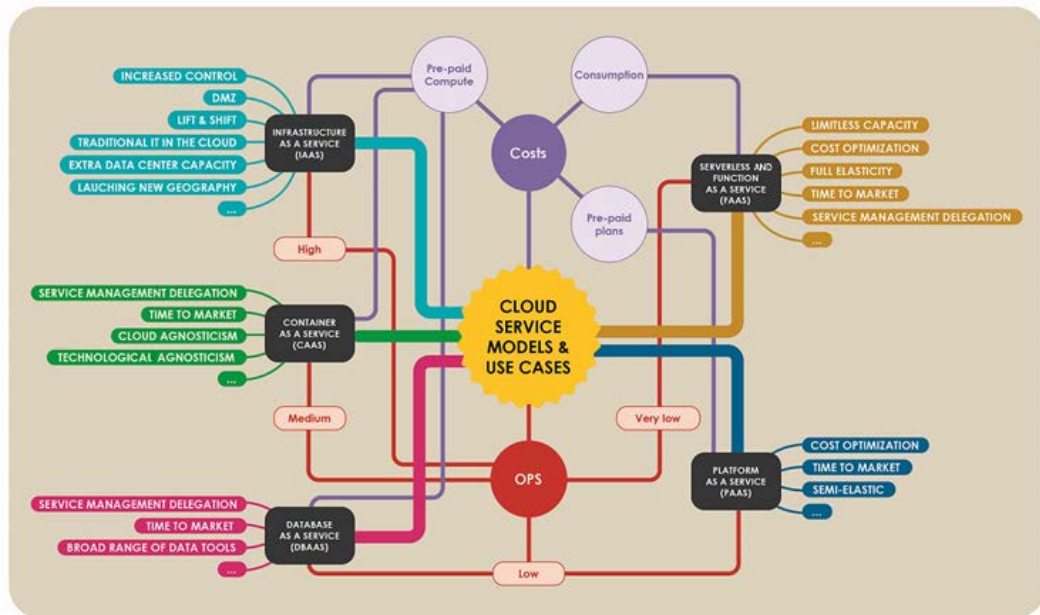


Figure 1.9 – Cloud service models

In terms of cost models, we see two big trends: **consumption** and **pre-paid compute/plans**. The consumption billing model is based on the actual consumption of dynamically allocated resources. Pre-paid plans guarantee a certain compute capacity that is immediately made available to the cloud consumer. In terms of operations, the map highlights what is done by the cloud provider, and what you still have to do yourself. For instance, **very low** means that you have almost nothing to do yourself. We will now walk through each service model.

IaaS (Infrastructure as a Service)

IaaS is probably the least disruptive model. It is basically the process of renting a data center from a cloud provider. It is business as usual (the most common scenario) in the cloud. IaaS is not the service model of choice to accomplish a digital transformation, but there are a number of scenarios that we can tackle with IaaS:

- The lift-and-shift of existing workloads to the cloud.
- IaaS is a good alternative for smaller companies that do not want to invest in their own data center.
- In the context of a disaster recovery strategy, when adding a cloud-based data center to your existing on-premises servers.
- When you are short on compute in your own data center(s).
- When launching a new geography (for which you do not already have a data center), and to inherit the cloud provider's compatibility with local regulations.
- To speed up the time to market, providing some legacy practices and processes were adjusted upfront to align with the cloud delivery model.

With regard to costs and operations, they are almost equivalent to on-premises, although it is very hard to compare the **total cost of ownership (TCO)** of IaaS versus on-premises.

Of course, facilities, physical access to the data center, and more are all managed by the cloud provider. It is no longer necessary to buy and manage the hardware and infrastructure software by yourself. However, you should be aware that most companies today have a hybrid strategy, which consists of keeping a certain number of assets on-premises, while gradually expanding their cloud footprint. In this context, IaaS is a required model to some extent, in order to bridge the on-premises and cloud worlds.

PaaS (Platform as a Service)

PaaS is a fully managed service model that helps you build new solutions (or refactor existing ones) much faster. PaaS reuses off-the-shelves services that already come with built-in functionalities and whose underlying infrastructure is fully outsourced to the cloud provider. PaaS is quite disruptive with regard to legacy systems and practices.

Unlike IaaS, in order to make a successful cloud journey, PaaS requires a heavy involvement from all the layers of the company and a top sponsor from the company's leadership. Make no mistake: this is a journey. With PaaS, much of the infrastructure and most operations are delegated to the cloud provider. The multi-tenant offerings are cost-friendly, and you can easily leverage the economies of scale, providing you adopt the PaaS model. PaaS is suitable for many scenarios:

- Green-field projects
- Internet-facing workloads
- The modernization of existing workloads
- API-driven architectures
- A mobile-first user experience
- An anytime-anywhere scenario, and on any device

The preceding list of use cases is not exhaustive, but it should give you an idea of what this service model's value proposition is.

FaaS (Function as a Service)

FaaS is also known as **serverless**. It emerged rather recently; it started with stateless functions that were executed on shared multi-tenant infrastructures. Nowadays, FaaS expanded to much more than just functions, and it is the most elastic flavor of cloud computing. While the infrastructure is also completely outsourced to the cloud provider, the associated costs are calculated based on the actual resource consumption (unlike PaaS, where the cloud consumer pre-pays a monthly fee based on a pricing tier). FaaS is ideal in numerous scenarios:

- **Event-driven architectures:** Receive event notifications and trigger activities accordingly. For example, having an Azure function being triggered by the arrival of a blob on Azure Blob Storage, parsing it, and notifying other processes about the current status of activities.
- **Messaging:** Azure Functions, Logic Apps, and even Event Grid can all be hooked to Azure Service Bus, handle upcoming messages, and, in turn, push their outcomes back to the bus.
- **Batch jobs:** You might trigger Azure Logic Apps or schedule Azure Functions to perform some jobs.
- **Asynchronous scenarios of all kinds**

- **Unpredictable system resource growth:** When you do not know in advance what the usage of your application is, but you do not want to invest too much in the underlying infrastructure, FaaS may help to absorb this sudden resource growth in a costly fashion.

FaaS allows cloud consumers to focus on building their applications without having to worry about system capacity, while still keeping an eye on costs. The price to pay for the flexibility and elasticity of FaaS is usually a little performance overhead that is caused by the dynamic allocation of system resources when needed, as well as less control over the network perimeter. This leads to some headaches for an internal **Security Operations Center (SOC)**, which is the reason why FaaS cannot be used for everything.

CaaS (Containers as a Service)

CaaS is between PaaS and IaaS. Containerization has become mainstream, and cloud providers could not miss that train. CaaS often involves more operations than PaaS. For example, **Azure Kubernetes Service (AKS)** involves frequent upgrades of the Kubernetes version on both the control plane and the worker nodes. Rebooting OS-patched worker nodes remains the duty of the cloud consumer. We could say that AKS is a semi-managed service as it is less managed than a PaaS or FaaS one, but it is much more managed than a regular IaaS virtual machine.

On the other hand, Web App for Containers is a fully managed service that sits between PaaS and CaaS, from a feature standpoint. **Azure Container Instances (ACI)** is also fully managed and sits between serverless (the consumption-pricing model) and CaaS. Admittedly, CaaS is probably the hardest model when it comes to evaluating both costs and the level of operations. It is, nevertheless, suitable for the following scenarios:

- **Lift-and-shift:** During a transition to the cloud, a company might want to simply lift and shift its assets, which means migrating them as containers. Most assets can be packaged as containers without the need to refactor them entirely.
- **Cloud-native workloads:** By leveraging the latest cutting-edge and top-notch Kubernetes features and add-ons.
- Batch, asynchronous, or compute-intensive tasks through ACIs.
- **Portability:** CaaS offers a greater portability, and helps to reduce the vendor lock-in risk to some extent.
- **Service meshes:** Most microservice architectures rely on service meshes. We will cover them in *Chapter 3, Infrastructure Design*.

- **Modern deployment:** CaaS uses modern deployment techniques, such as A/B testing, canary releases, and blue-green deployment. These techniques prevent and reduce downtime in general, through self-healing orchestrated containers.

We will explore the CaaS world in *Chapter 2, Solution Architecture*, and AKS in *Chapter 3, Infrastructure Design*.

DBaaS (Database as a Service)

DBaaS is a fully managed service model that exposes storage capabilities. Data stores, such as Azure SQL, Cosmos DB, and Storage Accounts, significantly reduce operations, while offering strong high availability and disaster recovery options. Other services, such as Databricks and Data Factory, do not strictly belong to the DBaaS category, but we will combine them for the sake of simplicity. Until very recently, Azure DBaaS was mostly based on pre-paid resource allocation, but Microsoft introduced the serverless model in order to have more elastic databases. DBaaS brings the following benefits:

- A reduced number of operations, since backups are automatically taken by the cloud provider
- Fast processing with Table Storage
- Potentially infinite scalability with Cosmos DB, providing that the proper engineering practices were taken up front
- Cost optimization, when the pricing model is well chosen and fits the scenario

We will explore DBaaS in *Chapter 6, Data Architecture*.

XaaS or *aaS (Anything as a Service)

Other service models exist, such as **IDaaS (Identity as a Service)**, to such an extent that the acronym **XaaS**, or ***aaS**, was born around 2016, to designate all the possible service models. It is important for an Azure architect to grasp these different models, as they serve different purposes, require different skills, and directly impact the cloud journey of a company.

Important note

We do not cover **Software as a Service (SaaS)** in this book. SaaS is a fully managed business suite of software that often relies on a cloud platform for its underlying infrastructure. SaaS examples include Salesforce and Adobe Creative Cloud, as well as Microsoft's own Office 365, Power BI, and Dynamics 365 (among others).

Now that we reviewed the most important service models, let's dive a little more into the rationale behind our maps.

Introducing Azure architecture maps

Although we have already presented a small map, let's explain how Azure architecture maps were born and how to make sense of them. However rich the official Microsoft documentation might be, most of it is textual and straight to the point, with walk-throughs and some reference architectures. While this type of information is necessary, it is quite hard to grasp the broader picture. An exception to this is the Azure Machine Learning Algorithm Cheat Sheet (<https://docs.microsoft.com/azure/machine-learning/algorithm-cheat-sheet>), which depicts, in a concise way, the different algorithms and their associated use cases. Unfortunately, Microsoft did not create cheat sheets for everything, nor is there any other real visual representation of the impressive Azure service catalog and its ecosystem. That gap leaves room for some creativity on the matter...and that is how **Azure architecture maps** were born. The primary purpose of Azure architecture maps is to help architects find their way in Azure, and to grasp, in the blink of an eye, the following:

- **Available services and components:** Since there are so many services and products out there, our primary purpose is to classify them and associate them with the most common customer concerns and use cases. However, keep in mind that Azure is a moving target! We will try to be as comprehensive as possible, but we can never guarantee exhaustive or complete coverage. It simply isn't possible.
- **Possible solutions:** These architecture maps are like a tree with multiple branches and sub-branches, and finally the branches end with flourishing leaves. On many occasions, there are multiple ways to tackle a single situation. That is why we will map alternative use cases, based on real-world experiences. However, we strongly encourage you to form your own opinion. You need to exercise your critical reflection on every topic, as to not blindly apply the map recommendations. The unique particularities of your own use case will often require a different solution (or at least a modified solution).
- **Sensitivity and trade-off points:** Architecting a solution is sometimes about choosing the lesser of two evils. Some of your choices or non-functional requirements might affect your final solution, or they might lead you to face some additional challenges. We will highlight sensitive trade-off points with specific marks on the maps.

Given the size of the Azure service catalog, a single map would not suffice. Hence, we created specialized maps. They are not restricted to Microsoft services and, when applicable, may also refer to marketplace and third-party solutions. Let's jump to the next section, which explains how to read and make sense of the maps.

How to read a map

The maps proposed in this book will be your Azure compass. It is therefore important to understand the fundamentals of how to read them. We will therefore go through a sample map to explain the semantics and its workings. *Figure 1.10* presents a very tiny, sample map:

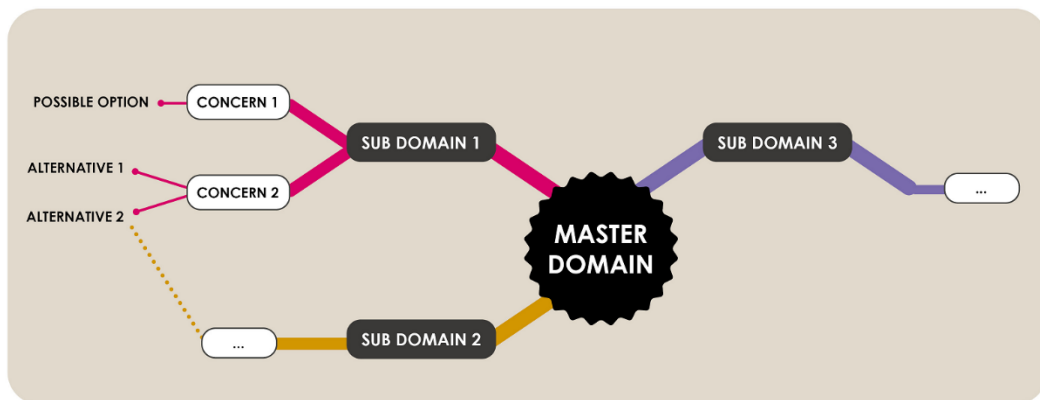


Figure 1.10 – A sample map

The central point that the diagram depicts is the **Master Domain (MD)**, the central topic of the map. Each branch represents a different area belonging to the MD. Under the sub domains, you can find the different concerns. Directly underneath the concerns, the different options (the tree's leaves) might help you address the concerns (see **POSSIBLE OPTION** in *Figure 1.10*). There might be more than one option to address for a given concern. For example, **CONCERN 2** in the diagram offers two options: **ALTERNATIVE 1** and **ALTERNATIVE 2**.

From time to time, dotted connections are established between concerns or options that belong to different areas, which indicates a close relationship. In the preceding example, we see that the **ALTERNATIVE 2** option connects down to the **SUB DOMAIN 2** concern. To give a concrete example of such a connection, we might find a **Dapr** leaf under the microservice architecture concern that is connected (by a dotted line) to a Logic Apps leaf under the integration concern. The rationale of this connection is because Dapr has a wrapper for self-hosted Logic Apps workflows. Let's now see how, as an architect, you can get started with your cloud journey.

Understanding the key factors of a successful cloud journey

The role of the Azure architect is to help enterprises leverage the cloud to achieve their goals. This implies that there is some preparation work up front, as there is no such thing as a one-size-fits-all cloud strategy. As we have just seen, the various cloud service models do not respond to the same needs and do not serve similar purposes. It is, therefore, very important to first define a vision that reflects which business and/or IT goals are pursued by your company before you start anything with the cloud.

As an example, typical transversal drivers (when moving to the cloud) are cost optimization and a faster time to market. Cost optimization can be achieved by leveraging the economies of scale from multi-tenant infrastructures. A faster time to market is conceived by maximizing outsourcing from the cloud provider. Should you have these two drivers in mind, rushing to a pure IaaS strategy would be an anti-pattern. Whatever your drivers, a possible recipe of success is the following: **Define Vision → Define Strategy → Start Implementation**. Let's now go through a few key aspects and start with the vision.

Defining the vision with the right stakeholders

Write a vision paper to identify what you are trying to solve with the cloud. Here are a few example questions for problems you might want to solve:

- Do you have pain points on-premises?
- Do you want to make data monetization through APIs?
- Do you want to outsource?
- Is the hardware in your data center at its end of life?
- Are you about to launch new digital services to a B2C audience?
- Do you have several of these issues?
- Are your competitors faster than you to launch new services to consumers, making you lose some market shares?

The vision paper helps you identify the business and IT drivers that serve as an input for your strategy.

Business drivers should come from the company's board of directors (or other corporate leaders). IT drivers should come from the IT leadership. Enterprise architecture may play a role in identifying both the IT and business drivers. Once the vision is clear for everyone, the main business and IT drivers should emerge and be the core of our strategy.

Defining the strategy with the right stakeholders

In order to achieve the vision, the strategy should be structured and organized around the vision. To ensure that you do not deviate from the vision, the strategy should include a cloud roadmap, cloud principles, and cloud governance. You should conduct a careful selection of candidate assets (greenfield, brownfield, and so on). Keep in mind that this will be a learning exercise too, so start small and grow over time, before you reach your cruising speed.

You should conduct a serious financial capability analysis. Most of the time, the cloud makes companies transition from CAPEX to OPEX, which is not always easy. You should see the cloud as a new platform. Some transversal budgets must be made available, to not be too tightly coupled to a single business project. Lastly, do not underestimate the organizational changes, as well as the impact of company culture on the cloud journey. Make sure that you integrate a change management practice as part of your strategy.

In terms of stakeholders, the extent to which the executive committee is involved should depend on the balance between business drivers and pure IT drivers. In order to be empowered to manage the different layers, the bare minimum requirement is to at least leverage a strong business sponsor. You should also involve the Chief Information Officer, or, even better, the Chief Digital Officer.

Starting implementation with the right stakeholders

This phase is the actual implementation of the strategy. Depending on the use case (such as a group platform), the implementation often starts with a scaffolding exercise. This consists of setting up the technical foundations (such as connectivity, identity, and so on). It is often a good idea to have a separate sandbox environment, to let teams experiment with the cloud. Do not default to your old habits, to using products you already use on-premises. Do your homework and analyze Azure's built-in capabilities. Only fall back to your usual tools after having assessed the cloud-native solutions. Stick to the strategy and the principles that were defined up front.

In terms of stakeholders, make sure you involve your application, security, and infrastructure architects (all together) from the start. Usually, the Azure journey starts by synchronizing Active Directory with Azure Active Directory for Office 365, which is performed by infrastructure teams. Since they start the cloud journey, infrastructure teams often tend to work on their own, and look at the cloud with infrastructure eyes only, without consulting the other stakeholders. Most of the time, this results in a clash between the different teams, which creates a lot of rework. Make sure that all the parties using the cloud are involved from the ground up, to avoid having a single perspective when designing your cloud platform.

The preceding advice is useful when building a cloud platform for a company. However, these factors are also often important to know for third-party suppliers, who would be engaged on a smaller **Request for Proposal (RFP)**. To deliver their solution, they might have to adhere to the broader platform design, and the sooner they know, the better.

Practical scenario

As stated in the previous sections, crafting a few principles that are signed off by the top management may represent a solid architecture artifact when engaging with various stakeholders in the company. Let's now go through a business scenario for which we will try to create an embryonic strategy:

Contoso is currently not using the cloud. They have all their assets hosted on-premises and these are managed in a traditional-IT way. The overall quality of their system is fine, but their consumer market (B2C) has drastically changed over the past 5 years. They used to be one of the market leaders, but competitors are now showing up and are acquiring a substantial market share year after year. Contoso's competitors are digital natives and do not have to deal with legacy systems and practices, which enables them to launch new products faster than Contoso, responding faster to consumer needs. Young households mostly use mobile channels and modern digital platforms, which is lacking in the Contoso offering. On top of this, Contoso would like to leverage artificial intelligence as a way to anticipate consumer behavior and develop tailor-made services that propose a unique customer experience by providing digital personal assistants to end users. However, while the business has some serious ambitions, IT is not able to deliver in a timely fashion. The business asked the IT department to conduct both an internal and external audit so as to understand the pain points and where they can improve.

Some facts emerging from the reports include (but are not limited to) the following:

- The adoption of modern technologies is very slow within Contoso.
- Infrastructure management relies entirely on the ITIL framework, but the existing processes and SLAs have not been reviewed for the past 5 years. They are no longer in line with the new requirements.
- The TCO is rather high at Contoso. The operational team headcount grows exponentially, while some highly qualified engineers leave the company to work in more modern environments.
- Some historical tools and platforms used by Contoso have reached end of life and are discontinued by vendors in favor of their corresponding cloud counterpart, which made Contoso opt for different on-premises solutions, leading to integration challenges with the existing landscape.

As a potential solution, the auditors proposed a magical recipe: the cloud (Azure in our case)! Now, it's up to you, the Azure architect, to manage expectations and advise Contoso on the next steps. We will see an example of this work in the next sections.

The drivers

Some drivers emerge rather quickly out of this business scenario. The business wants to launch products faster, so time to market is critical. Costs are never mentioned by the business, but the audit reveals a **TCO** increase due to growing operational teams. So, costs are not a strong focus, but we should keep an eye on it. The features the business want to expose as part of their services rely on top-notch technologies, which are hard to make available on-premises. So, technology could be a business enabler for Contoso. In summary, the drivers that emerge are **time to market**, **new capabilities** (enabled by top-notch technologies), and, to a lesser extent, **cost optimization**.

Strategy

We could write an entire book on how to conduct a proper strategy, so we will simplify the exercise and give you some keys to get started with your strategy. To understand all the aspects that you have to keep an eye on, you can look at the Microsoft Cloud Adoption Framework (<https://docs.microsoft.com/azure/cloud-adoption-framework/>). This is a very good source of information, since it depicts all the aspects to consider when building an Azure cloud platform. To structure and formalize your strategy, you could also leverage governance frameworks such as **Control Objectives for Information and Related Technologies (COBIT)** (<https://www.isaca.org/resources/cobit>). This helps transform verbal intentions into a well-documented strategy, and to consolidate the different aspects so as to present them in front of executive people. It also connects the dots between the business goals and the IT goals in a tangible fashion. One of the key COBIT artifacts is what they call *the seven enablers*, which are applicable to any governance/strategy plan:

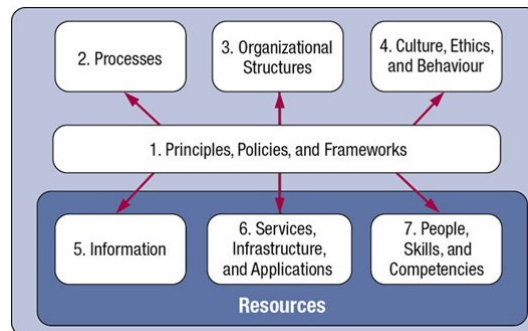


Figure 1.11 – Cobit's seven enablers

The diagram offers a short definition of each, and their relative impact on the journey. You can easily map them to the dimensions you see in the CAF:

1. **Principles, Policies, and Frameworks:** This could be summarized as such: *what is clearly thought is clearly expressed*. You should identify your core principles and policies that are in line with the business drivers. These will be later shared and reused by all involved parties. Writing a mission statement is also something that may help everyone to understand the big picture.
2. **Processes:** The actual means to executing policies and transforming the principles into tangible outcomes.
3. **Organizational Structures:** A key enabler to putting the organization in motion toward the business and IT goals. This is where management and sponsorship play an important role. Defining a team (or virtual cloud team), a stakeholder map, and, first and foremost, a platform owner, accountable for everything that happens in the cloud, who can steer activities.
4. **Culture, Ethics, and Behavior:** This is the DNA of the company. Is it a risk-adverse company? Are they early adopters? The mindset of people working in the company has a serious impact on the journey. Sometimes, the DNA is even inherited from the industry (banking, military, and so on) that the company operates in. With a bit of experience, it is easy to anticipate most of the obstacles you will be facing just by knowing the industry practices.
5. **Information:** This enabler leverages information practices as a way to spread new practices in a more efficient way.
6. **Services, Infrastructure, and Applications:** Designing and defining services is not an easy thing. It is important to re-think your processes and services to be more cloud-native, and not just lift and shift them as is.
7. **People, Skills, and Competencies:** Skills are always a problem when you start a cloud journey. You might rely on different sourcing strategies: in-staffing, outsourcing, . . . , but overall, you should always try to answer the question everyone is asking themselves: *what's in it for me in that cloud journey?* In large organizations, a real change management program is required to accompany people on that journey.

Developing a strategy around all these enablers is beyond the scope of this book. From our real-world experience, we can say that you should work on all of them, and you should not underestimate the organizational impacts and the cultural aspects, as they can be key enablers or disablers should you neglect them. A cloud journey is not only about technology; that's probably even the easiest aspect.

To develop our strategy a little further, let's start with some principles that should help meet the business drivers expressed by Contoso, for whom **time to market** is the most important one:

- **SaaS over FaaS over PaaS over CaaS over IaaS:** In a nutshell, this principle means that we should *buy* instead of *building* first, since it is usually faster to buy than build. If we build, we should start from the most provider-managed service model to the most self-managed flavor. Here again, the idea is to gain time by delegating most of the infrastructure and operational burden to the provider, which does not mean that there is nothing left to do as a cloud consumer. This should help address both the time-to-market driver, as well as the exponential growth of the operational teams. From left to right, the service models are also ordered from the most to the least cloud-native. CaaS is an exception to this, but the level of operational work remains quite important, which could play against our main driver here.
- **Best of suite versus best of breed:** This principle aims at forcing people to first check what is native to the platform before bringing their own solutions. Bringing on-premises solutions to the cloud inevitably impacts time. Best of suite ensures a higher compatibility and integration with the rest of the ecosystem. Following this principle will surely lock you in more to the cloud provider, but leveraging built-in solutions is more cost- and time-efficient.
- **Aim at multi-cloud but start with one:** In the longer run, aim at multi-cloud to avoid vendor locking. However, start with one cloud. The journey will already be difficult, so it's important to concentrate the efforts and stay focused on the objectives. In the short term, try to make smart choices that do not impact cost and time: do not miss low hanging fruit.
- **Design with security in mind:** This principle should always be applied, even on-premises, but the cloud makes it a primary concern. With this principle, you should make sure to involve all the security stakeholders from the start, so as to avoid any unpleasant surprises.
- **Leverage automation:** Launching faster means having an efficient CI/CD toolchain. The cloud offers unique infrastructure-as-code capabilities that help deploy faster.
- **Multi-tenant over privatization:** While privatization might give you more control, it also means a risk of reintroducing your on-premises practices to the cloud. Given the audit reports we had, we see that this might not be a good idea. Leveraging multi-tenant PaaS services that have been designed for millions of organizations worldwide is a better response to the business drivers.

This is not necessarily where the list ends. Other principles could be created.

Having different drivers would give us different principles. The most important thing is to have concise, self-explicit, and straightforward principles. Now that you have this first piece done, you can build on it to further develop your policies and the rest of your strategy. This will not be covered in this book, but you had a glimpse of what a cloud strategy looks like. So, do work on this in your own time. The time has now come to recap this chapter.

Summary

In this chapter, we reviewed the architecture landscape and the different types of architects we may be working with in our day-to-day Azure architecture practice. Knowing the different profiles, being able to speak to each of them, as well as satisfying their own interests and preoccupations, is what every Azure architect should do.

In this chapter, we also explained the value proposal of the maps and how to read a map, which will be very useful for the next chapters. We shed some light on the various service models that exist in the cloud, and those that serve different purposes. We also tried to grasp the important differences that exist across them, in terms of functionalities, operations, and costs. All these models constitute the cornerstone of Azure, and should be wholly mastered by the Azure architect as they represent the minimal, vital must-have skills. Finally, we have understood the key success factors of a cloud journey from real-world observations and through a fictitious enterprise scenario.

In the next chapter, we will start to get closer to the actual implementation of an Azure-based solution.

2

Solution Architecture

In this chapter, we will review the broad landscape of Azure, looking at it through the eyes of a solution architect. More specifically, we will cover the following topics:

- The solution architecture map
- Zooming in on the different workload types
- Zooming in on containerization
- Solution architecture use case

This chapter should give you the keys to designing any type of solution with Azure. After reading this chapter, you should be able to find a workable solution, whatever use case you are confronted with. However, you still need to refer to the other maps of this book in order to dig deeper into additional in-depth information.

Technical requirements

Admittedly, pure architecture is not typically hands-on. However, by the end of the chapter, we will have walked you through the design of a reference architecture and its corresponding code implementation. To open the physical reference architecture file and test the code, you will need the following:

- Visual Studio 2019 to run and debug the provided .NET Core program locally. It makes use of a local Azure Storage emulator and the Azure Functions runtime.
- Fiddler, Postman, or any HTTP tool you want, to send HTTP requests to our program.
- Microsoft Visio to open the diagrams. We also provide the corresponding PNG files.

The full code files are available at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/tree/master/Chapter02>.

The CiA videos for this book can be viewed at: <http://bit.ly/3pp9vIH>

The solution architecture map

The purpose of the **solution architecture map** is to help solution architects find their way in Azure. We defined the duties of a solution architect in the previous chapter, which is typically an architect who assembles the different building blocks and services of a solution while considering the non-functional requirements. Solution architects engage with their specialized peers, who are often application, infrastructure, and security architects.

This solution architecture map, illustrated in *Figure 2.1*, does not alone regroup all the Azure services, nor does it cover all the possible use cases, but it can be used as a source of inspiration. Depending on the extent to which you want to practice solution architecture, we suggest that you carefully read every chapter of this book, to be able to consider the end-to-end aspects of a solution:

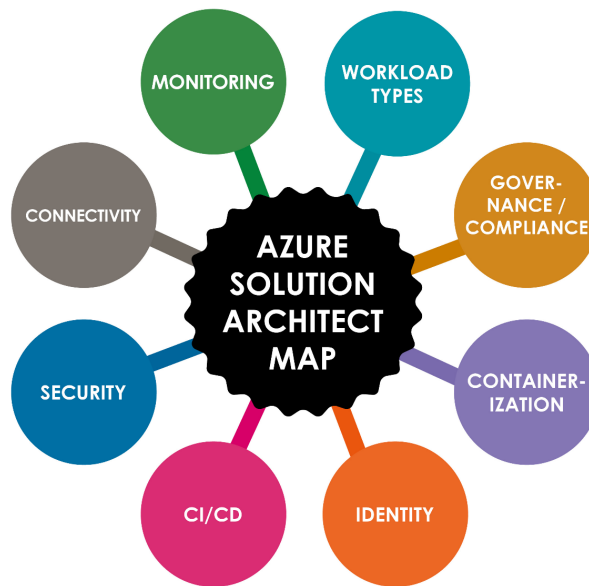


Figure 2.1 – The solution architecture map

Important note

To see the full solution architecture map (*Figure 2.1*), you can download the PDF file that is available here: <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/blob/master/Chapter02/maps/Azure%20Solution%20Architect%20Map.pdf>. You can use the zoom-in and zoom-out features to get a better view.

In *Figure 2.1*, the **Azure Solution Architect Map** starburst acts as the hub of a wheel. There are eight spokes of the wheel:

- **MONITORING**
- **WORKLOAD TYPES**
- **GOVERNANCE / COMPLIANCE**
- **CONTAINERIZATION**
- **IDENTITY**
- **CI/CD**
- **SECURITY**
- **CONNECTIVITY**

Later in this chapter, we will zoom in on the different spokes (areas) of the solution architecture map, but first, we're going to focus on the different workload types that span across this architecture.

Zooming in on the different workload types

In the following sections, we will take a closer look at some typical use cases and cross-cutting concerns that solution architects deal with (we refer to these workload types as **categories**). Separating the map into smaller sections makes it easier to digest and understand. Let's begin by looking at **Systems of Engagement (SoE)**.

Understanding systems of engagement

This category regroups all the services that belong to the frontend layer, such as user interfaces, mobile apps, and every channel that helps engage with first and third parties:

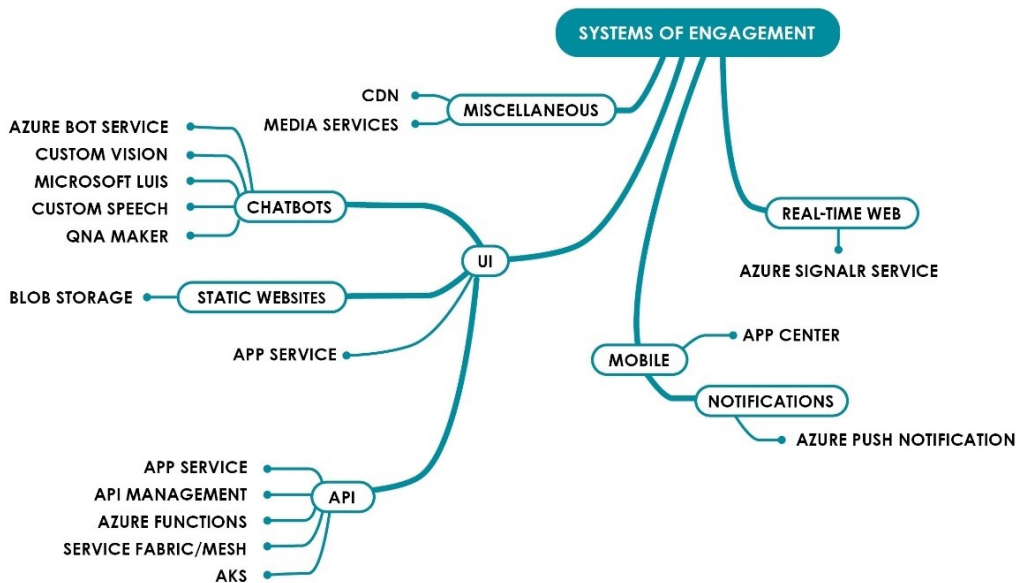


Figure 2.2 – The SoE category

In *Figure 2.2*, the SoE category map includes four top-level groups:

- **REAL-TIME WEB**
- **MOBILE**
- **UI**
- **MISCELLANEOUS**

The miscellaneous group includes CDN and Media Services. **Azure Content Delivery Network (CDN)** helps you reduce your load times for more responsive customer interactions. If you are working with rich media and video streaming, **Azure Media Services** should be your primary choice.

The next group in *Figure 2.2* is UI. First, let's discuss the sub-group of chatbots. **Chatbots** have become popular because of their user-friendliness. Azure has a lot to offer in that area. **Azure Bot Service** integrates with many channels (Alexa, Cortana, Direct Line, Facebook, Microsoft Teams, Slack, and so on) to increase the reach of your chatbots. **QnA Maker** and **LUIS (Language Understanding Intelligent Service)** help drive conversations, and **Azure Cognitive Services** facilitates richer interactions through audio and visual features (QnA Maker and LUIS are features in Cognitive Services). In our chatbots list (in *Figure 2.2*), we specifically call out the Cognitive Services APIs: Custom Vision, which tags images and extracts text from them, and Custom Speech, which can turn speech into text (such as voice commands for your app).

Under the UI group (in the map of our SoE category), we also have static websites, Azure App Service, and the API sub-group. **STATIC WEBSITES** points to **AZURE BLOB STORAGE**, which is a storage type where you can place all kinds of files (in this case, you would have built your website using HTML and other files, and you upload your files into Blob storage). It's called static because you're not dynamically changing your website with Azure (you'd have to go in and replace the files).

Next, **Azure App Service** is a fully managed, first-class citizen for hosting frontend applications, **Backends for Frontends (BFF)**, and backend services in general. It supports a wide set of programming languages (at the time of writing, this includes .NET, .NET Core, Java, Node.js, PHP, Python, and Ruby), and it is available over multiple pricing tiers, which range from multi-tenant to isolated. App Service can easily be integrated with a deployment factory for **Continuous Integration and Continuous Deployment (CI/CD)** purposes, and it allows zero-downtime through the use of deployment slots. It is also ideal for the following scenarios:

- MVC web applications
- API services
- Lift and shift of legacy .NET applications, by using Web App for Containers with Windows
- Pre-container-orchestration scenarios, by using Web App for Containers with Linux or Windows

App Service can also be used with pure **Single-Page Applications (SPAs)**. However, the current trend is to offload all the static files (such as `.js`, `.css`, `.png`, and so on) to a storage account that is proxied by Azure CDN or Azure Front Door for optimized speed. App Service relies on **App Service plans**, which define the compute that is allocated to the service(s). A plan may host one or more services and support both horizontal and vertical (auto)scaling, either for the entire set of apps or through a per-app horizontal scaling feature.

That brings us to the **API** sub-group (under **UI**). When used as a BFF, App Service should be proxied by an **API gateway**, in order to enhance the overall security for the instance, by validating the **JWTs (JSON Web Tokens)** or client certificates, as well as to offload typical API concerns, such as throttling, caching, and so on. App Service is mature, robust, and easy to use, with minimal operations. Of course, Azure App Service is not the only option to host a BFF. A BFF can also be hosted in **Azure Kubernetes Service (AKS)** or Service Fabric Mesh. The final service in the API sub-group is Azure Functions, which are small blocks of code that respond to messages, requests, or schedules (such as backing up or archiving customer data), or can be part of a broader microservices architecture.

Next, let's move on to the **MOBILE** top-level group (in our SoE category in *Figure 2.2*). Regarding mobile scenarios, we can still leverage Azure App Service as a BFF and **Azure Notification Hubs** in order to send push notifications to both mobile and non-mobile apps whenever we see fit. **Microsoft App Center** (also called the **Visual Studio App Center**) is handy to deploy and test mobile apps. It used to have a push notification feature, but this was retired in December 2020.

The final top-level group is **REAL-TIME WEB**. End users often request the ability to be notified when a task completes, or to watch changes in near real time as tasks occur. A common example of this is a chat application. This can be achieved with **Azure SignalR Service**, which offloads the SignalR backend to Azure, instead of requiring you to host your own SignalR server.

Understanding systems of record

Strictly speaking, **Systems of Record (SoR)** refers to databases and data integrity. An SoR might (or might not) be transactional or mission-critical:

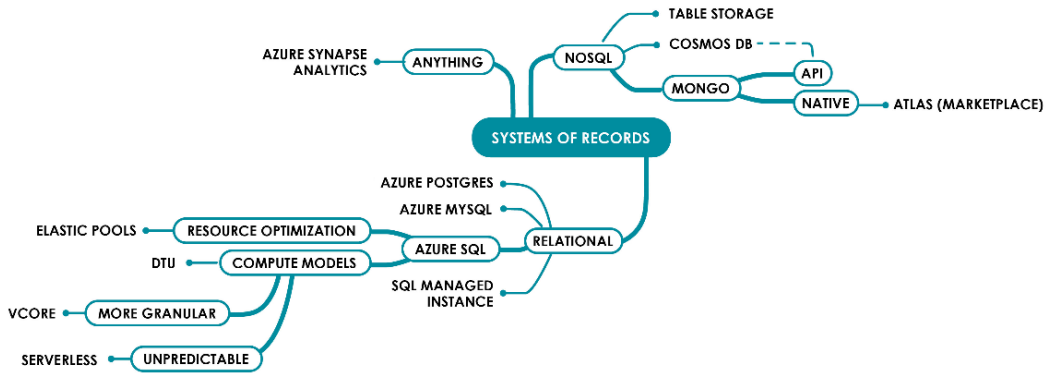


Figure 2.3 – The SoR category

For our SoR category map (*Figure 2.3*), we have just three top-level groups:

- **ANYTHING**
- **RELATIONAL**
- **NOSQL**

From the **ANYTHING** group, we'll unfold **Azure Synapse Analytics** (formerly known as **Azure Synapse**) more later in this chapter, but this is where you can get big data analytics against large enterprise data warehousing.

That brings us to the next two groups. We can distinguish between these two main storage types: the **RELATIONAL** and **NOSQL** data stores. The rationale of using a NoSQL engine versus a SQL one is beyond the scope of this book but we can try to summarize the main reasons that lead you to choose either of these. NoSQL systems are designed to scale, and mostly rely on the **BASE (Basically Available, Soft State, Eventual Consistency)** model, whose biggest impact is **eventual consistency**. Their purpose is to respond fast to read requests, at the cost of data accuracy. They are suitable for **big data**. Some variants of NoSQL implementations, such as Azure Table storage, also embrace strong consistency, but will not scale as much as an eventual consistency-based Cosmos DB. Traditional SQL engines rely on the **ACID (Atomicity, Consistency, Isolation, Durability)** model, which is centered around transactions and data accuracy, at the cost of speed. They are nothing new, since we have used them for decades, but it is worth mentioning that Azure SQL Database ships with two compute models: DTU and vCore. The DTU model is a combination of CPU and memory resources that are allocated to a single database or an elastic pool. This compute model aims at simplifying cost calculations, but it is rather obscure compared to vCore. The latter is more transparent, since you see exactly what is charged for the CPU and memory. You can scale that differently, while DTU regroups everything together.

A way to optimize database resources is to use SQL Database **elastic pools**, which allocate resources dynamically, based on the actual needs of the databases. Whenever you deal with sporadic resource-intensive workloads, you should consider the **Azure SQL serverless compute tier**. (It automatically scales compute, based on the workload demand, and it bills for the amount of compute that's used, per second.) However, this is only valuable when dealing with databases that have important idle times. The reason is that the allocated compute is only deallocated after an hour of inactivity, which could then lead to higher costs if the database never idles for more than an hour. Therefore (this is inspired by a true story), you should avoid running availability/advanced tests that end up making a call to such a database.

Finally, we arrive at the NoSQL group (in *Figure 2.3*). Cosmos DB is Azure's best NoSQL storage solution. It currently (as of the time that this book was written) supports five API models: SQL, Mongo, Table, Cassandra, and Gremlin. The native MongoDB API is only available by relying on Atlas, a marketplace product, or by self-hosting a MongoDB, which in this case is often combined with AKS. Cosmos DB's preferred API model is SQL, as it offers the best performance, so you should try to favor this option first. Going for another model might be driven by a constraint such as making a lift and shift of an existing MongoDB to Cosmos, or simply a technology preference.

Other types of data stores (such as Blob storage, Redis cache, and services for data in transit) do not exactly correspond to the SoR definition. However, be aware that there are many more services that can store data in one way or another. We will cover them later on in this book.

Understanding systems of insight

This category regroups data analysis services, helping to extract valuable business insights out of both SoR and SoE, but it also does so from any type of data store:

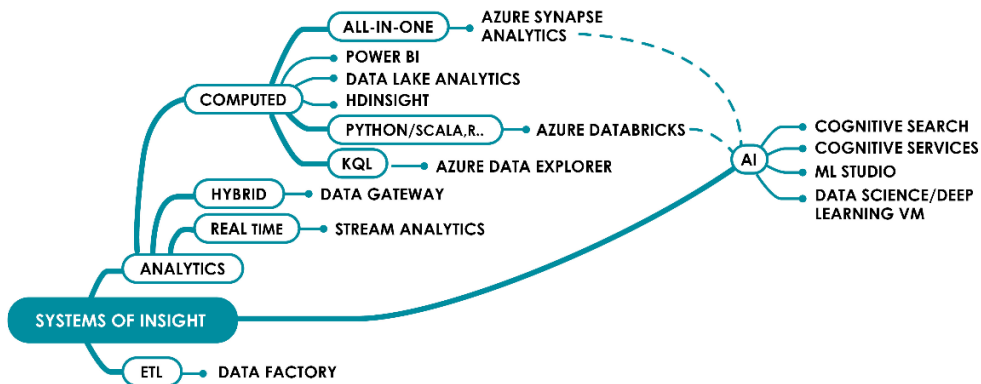


Figure 2.4 – The SoI category

In our **Systems of Insight (SoI)** category map (*Figure 2.4*), we have the following top-level groups:

- **ANALYTICS**
- **ETL**

Each top-level group is divided into subgroups that we are going to explain as follows.

Let's start at the top and look at the **AI** sub-group. For predefined AI services, you can rely on **Azure Cognitive Services**, a set of APIs that help when dealing with **Optical Character Recognition (OCR)**, image tags, **Natural Language Processing (NLP)**, and so on. Some of these APIs are mainstream and non-customizable, while others, such as LUIS and the Custom Vision service, can be easily trained to build models that are tailor-made to meet your needs. Cognitive Search enables you to leverage the built-in AI capabilities to identify relevant content at scale.

Machine Learning Studio (the web portal for Azure Machine Learning) comes with typical algorithms for which Microsoft has built the machine learning cheat sheet (<https://docs.microsoft.com/azure/machine-learning/algorithm-cheat-sheet>), which we mentioned in *Chapter 1, Getting Started as an Azure Architect*.

Machine Learning Studio is a fully managed service. It has some very nice features that allow you to test multiple models in parallel against the same set of training data, as well as to evaluate them all together in order to see which one performs the best. Contrarily, deep learning virtual machines are self-managed, but they are preconfigured with all the popular AI frameworks. Azure Data Explorer and Azure's native query language for many services is the **Kusto Query Language (KQL)**, which is mostly used in monitoring and log analysis.

Now, let's move down to our **COMPUTED** sub-group (from our SoI category map, in *Figure 2.4*). **Azure Synapse Analytics** is Azure Data Warehouse's successor, but this is more than just a rebranding! Azure Synapse glues many data services together (Azure Data Warehouse, Data Lake, Power BI, Spark clusters, Azure Machine Learning, and so on) in order to analyze both enterprise and big data. It is intended to be used by both data scientists and traditional **Business Intelligence (BI)** analysts in a single consolidated service.

Power BI is a comprehensive service that allows you to create reports and dashboards (including real-time dashboarding) at an enterprise scale. Power BI's real-time dashboards can be very handy in any **Business Activity Monitoring (BAM)** scenario. Azure Data Lake Analytics processes big data jobs (petabytes of data) in seconds, and **Azure HDInsight** is an easy and cost-effective method for running open source analytics, such as **Apache Hadoop, Spark, and Kafka**.

Azure Databricks is a very comprehensive service that you can use to analyze data at scale. It relies on a cluster and can be fed by any data source. It encompasses **Artificial Intelligence (AI)** by leveraging data science frameworks. Databricks requires very specific skills, although the SQL language is accessible to non-data scientists. **Azure Data Explorer** helps non-data scientists extract useful insights by using KQL.

In our **HYBRID** sub-group, we find the **data gateway**, which acts as a bridge between the cloud and on-premises data sources. The gateway comes in two flavors: personal and shared. The personal gateway allows users to create Power BI reports against on-premises data, for their own use. The shared gateway is a cross-user and cross-service gateway.

Stream Analytics is a serverless service for *real-time analytics*. It is often used in conjunction with Power BI real-time dashboards for BAM and IoT scenarios. It also helps make decisions on data as it flows into your system. We will zoom in deeper on the data services in *Chapter 6, Data Architecture*.

Finally, in the **ETL (or ELT)** sub-group (at the bottom of *Figure 2.4*), **Azure Data Factory** is another fully managed service that can be used for both ETL and ELT purposes. It can be combined with Databricks notebooks and Azure Functions. Pipelines can be triggered through API calls, and they can react to Azure Event Grid notifications.

Understanding systems of interaction (IPaaS)

Systems of interaction represent the way to integrate SoE with SoR, as well as to integrate with other systems. In Azure, we can refer to this as **Integration Platform as a Service (IPaaS)**. This category regroups services that enable integration between different layers of a single solution or across multiple solutions. As usual, *Figure 2.5* is available as PNG and VSDX files in our GitHub repo:

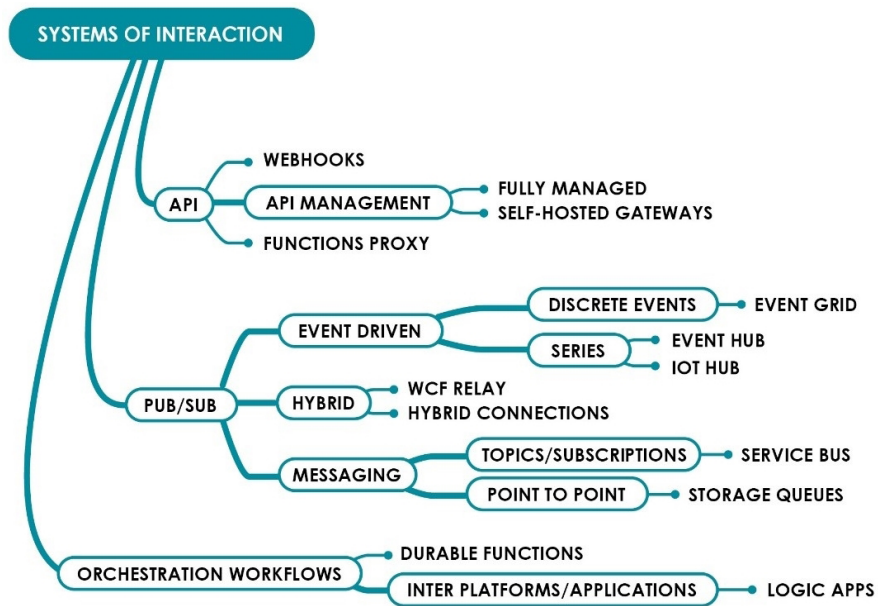


Figure 2.5 – The IPaaS category

In the systems of interaction (IPaaS) category map (*Figure 2.5*), we have three top-level groups:

- **API**
- **PUB/SUB**
- **ORCHESTRATION WORKFLOWS**

In the **API** top-level group, we can use **Azure Functions**, which has many built-in bindings that interact with many different stores. Azure Functions ships on the consumption tier (serverless) and pre-paid tiers through an **Azure App Service plan**. It can also be self-hosted using containers, in which case you should look at **Kubernetes Event-Driven Autoscaler (KEDA)** to handle the scaling aspects. You might wonder why you would self-host functions, instead of letting Microsoft do so. *Figure 2.6* is another focused map that will help you understand some important differences:

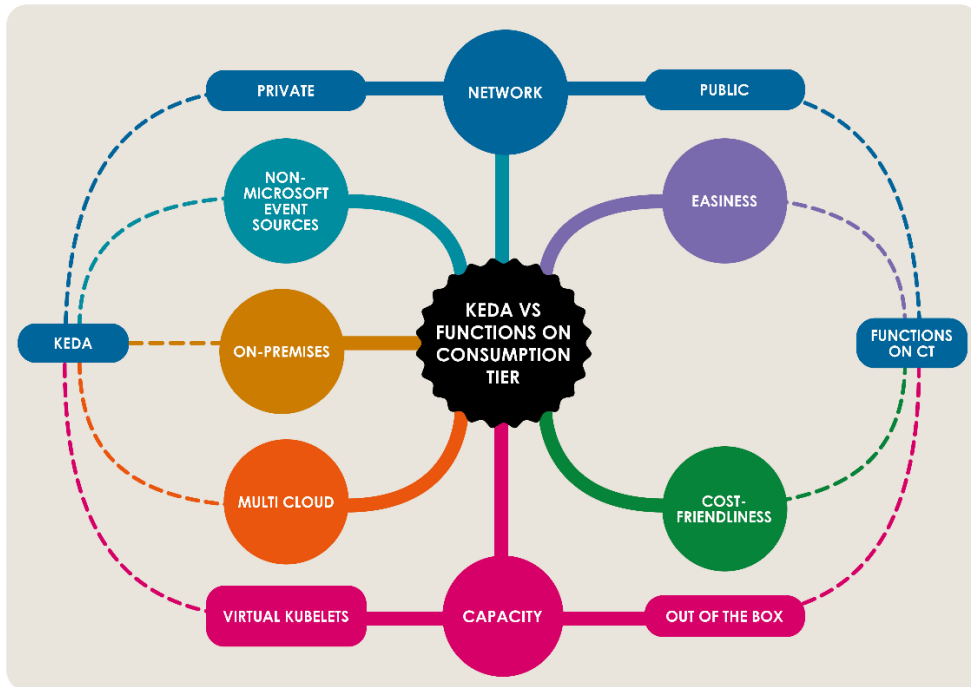


Figure 2.6 – KEDA versus the consumption tier of Azure functions

The typical reason is private connectivity. Whenever you want fully private connectivity, for both your function endpoints and the stores they talk to, self-hosting functions are often required. There is an alternative that consists of using the premium tier of Azure Functions, but this is rather expensive, and it does not yet offer the same level of network isolation. KEDA brings the intelligence of Azure Functions to AKS, by playing the man in the middle between the event stores and the handlers. KEDA also integrates with many more stores, including non-Microsoft cloud platforms, such as AWS, Alibaba, and so on. That is one of the low-hanging fruits we discussed in *Chapter 1, Getting Started as an Azure Architect*, where we advised you to make smart choices to avoid vendor-locking while not starting multiple clouds at once.

Beyond Azure Functions, you can also rely on **Azure Container Instances (ACI)**, **Azure Automation runbooks**, and basically any API through **webhook** integration.

Nowadays, integration is made through APIs. Azure **API Management (APIM)** is Azure's first-class citizen for exposing APIs to other systems and organizations. It allows you to manage your APIs through many features. The following list highlights the main APIM features:

- **Versioning:** Dealing with multiple versions of the same API, for backward compatibility.
- **Revisions:** Testing API changes, with the ability to promote them later on, as the real/deployed versions.
- **Products:** Clubbing APIs together into a product, and then letting API consumers subscribe to the product.
- **Policies:** Enforcing controls, such as JWT token validation, throttling, HTTP header check, request/response transformations, and so on. Policies are enforced by the API gateway, which is also known as the **Policy Enforcement Point (PEP)**. API gateways can also be self-hosted, mostly in hybrid scenarios (on-premises, cloud, or cloud-to-cloud).
- **Developer portal:** Letting consumers discover and subscribe to your APIs.
- **Publisher portal:** Managing your APIs.

The preceding list is not exhaustive, but the key thing to remember is that all API management systems (not only Azure's APIM) play an important role when integrating different systems together. APIM is mostly used in **Business-to-Business (B2B)** contexts, when selling APIs to other parties, but it's also used in microservices architectures. Logic Apps can be used to consume APIs that are made available by APIM.

Azure has native services to deal with all types of integration. In the publish/subscribe domain (see the **PUB/SUB** top-level group in *Figure 2.5*), two services emerge: **Azure Service Bus** and **Azure Event Grid**. Service Bus is mostly used in messaging scenarios, while Event Grid is suitable in **Event-Driven Architectures (EDAs)**. The boundaries between messaging and EDA are sometimes blurred, because an event itself is a message. An event is generally used to tell others that something happened, while a message instructs others to act on it. *Figure 2.7* depicts Azure's EDA landscape (as usual, you can find the JPG and VSDX files for the EDA map in our GitHub repo):

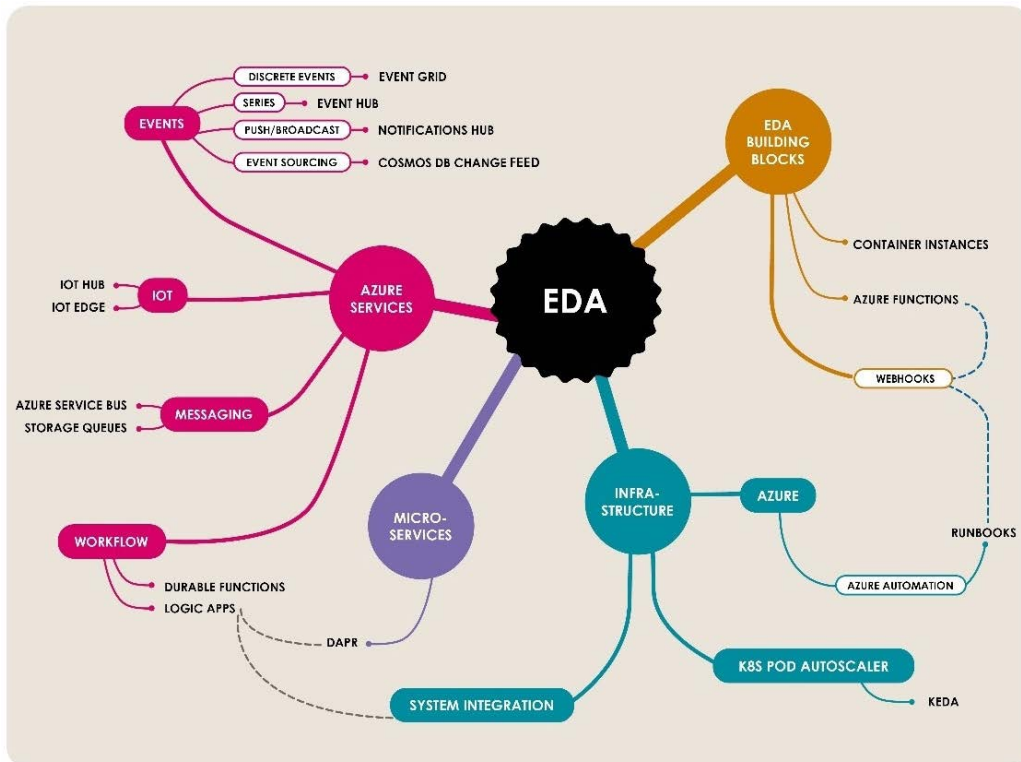


Figure 2.7 – A map focused on EDA and messaging architectures

We see that Event Grid is more suited for discrete events, which are independent events that reflect a state change. Event Grid can be used to notify receiving parties. In contrast, **Azure Event Hubs** is ideal for a large sequence of correlated events, such as for logging metrics of API calls, further analysis, and fine-grained billing.

Azure Queue Storage is mostly used for point-to-point message exchange and the queue-based load leveling pattern, which is also possible with **Service Bus queues**. In such a situation, where you simply need a mere queue, instead of a complete pub/sub, you might look at the pricing to make your choice.

Finally, let's dig into the third top-level group of our IPaaS map, orchestration workflows (see the bottom of *Figure 2.5*).

For these workflow-like workloads, **Azure Durable Functions** and **Logic Apps** are the main services you would use. The main difference between them is related to integration connectors that ship with Logic Apps. Later in this chapter, you will find *Figure 2.20* that highlights the differences between them, but we can already say that Logic Apps is Azure's star product for any integration scenario. Let's now address some typical cross-cutting concerns.

Looking at cross-cutting concerns and non-functional requirements

Building a solution is not only about programming. You may have the best developers ever, producing the best code, and you still might end up with a very poor customer experience. Beyond the workload classification itself, you need to look at the cross-cutting concerns and non-functional requirements, which all contribute to a production-grade application.

But what exactly is a production-grade application? It can be defined as an application that meets the following qualifications:

- Performs well
- Is reliable and secure
- Can be deployed easily and frequently
- Is hosted on a resilient system
- Is hosted in a governed system, a landing zone for your cloud solutions
- Is properly monitored
- Has a clear service model defined, such as detailing who does what when a problem occurs

Most of the previous items only partially depend on the code. This is why we shed some light on these transversal concerns in the next sections. The following sections are the zoom-in maps, which show us detailed views of the spokes of this chapter's solution architecture map (*Figure 2.1*).

Learning about monitoring

Monitoring is mandatory for any system that runs in production. The purpose of monitoring systems and applications is to detect adverse events as soon as possible, and then to react either manually or automatically. Monitoring also makes it possible to have global oversight of the running systems. You can then collect **Key Performance Indicators (KPIs)** to evaluate the service quality toward your proposed **Service-Level Agreements (SLAs)**. See *Figure 2.8* to zoom in on the monitoring category:

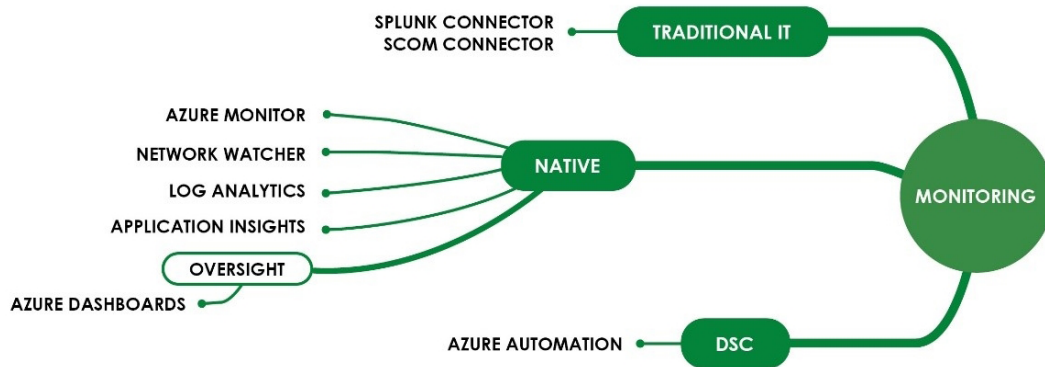


Figure 2.8 – Zoom in on monitoring

In *Figure 2.8*, we have three top-level groups: **TRADITIONAL IT**, **NATIVE**, and **DSC**. Let's discuss the traditional IT group. Azure's native monitoring tools are quite robust, but you're sometimes required to integrate your system with products that are typically used for on-premises systems, such as Splunk. The good news is that most of these products ship with built-in Azure connectors.

The same thing applies to IBM QRadar for security logs. Depending on the situation, it's a viable option to use Azure-only monitoring. Likewise, **SCOM CONNECTOR** (in the traditional IT group) refers to an Azure connector that enables you to connect to Microsoft **System Center Operations Manager (SCOM)**. In any case, we would recommend that you leverage native services as much as possible, even when on-premises integration is required. The options are not mutually exclusive, although log ingestion comes at a cost.

Now, let's go to the **NATIVE** top-level group. **Azure Monitor** is the main service that collects all the metrics, while **Log Analytics** (an Azure Monitor feature) is usually used as a central log repository to query logs through KQL. You can define alerts on both metrics and logs. Alerts are sent through action groups, which range from sending emails and SMS to triggering automated responses via **Logic Apps**, which integrates with virtually anything. **Azure Network Watcher** is a set of tools for you to diagnose and enable/disable resource logs in an Azure virtual network. Application Insights is a feature of Azure Monitor, which works as a performance management service for developers. Azure dashboards are useful to quickly view a visual representation of a solution's health. A good practice is to leverage **Infrastructure as Code (IaC)** with pre-defined dashboards for every deployed application.

Lastly, in our **DSC** group (see *Figure 2.8*), **Azure Automation** is the only native Azure service that ensures a **Desired State Configuration (DSC)**, which targets virtual machines. DSC consists of defining a certain configuration and getting the service to auto-correct deviations. Azure Automation meets similar needs as Ansible, Chef, Puppet, and so on. Azure Automation can also be used to provision Azure Resources through Automation runbooks, either directly or from CI/CD pipelines through webhooks. Its hybrid workers also make it easy to interact with on-premises systems.

From the very beginning, an Azure solution architect should envision a monitoring strategy that is scoped to either a single solution or that is scoped to integrate with a strategy that is defined for the entire platform. This will impact deadlines if a monitoring strategy is not considered from the start. If a requirement from the landing zone is to redirect logs to Azure Event Hubs, then in order to let Splunk ingest them on-premises, you, the Azure solution architect, must supervise the activities accordingly, and reflect this in an initial set of solution diagrams.

Learning about factories (CI/CD)

In theory, CI/CD has become mainstream. However, we often see big gaps and a lack of understanding in that matter. Since leveraging CI/CD to its maximum extent has an impact on how applications are built and deployed, it is important for the solution architect to assess the level of readiness of the existing CI/CD factory (if any). *Figure 2.9* zooms in on CI/CD spoke/area of our solution architecture map:

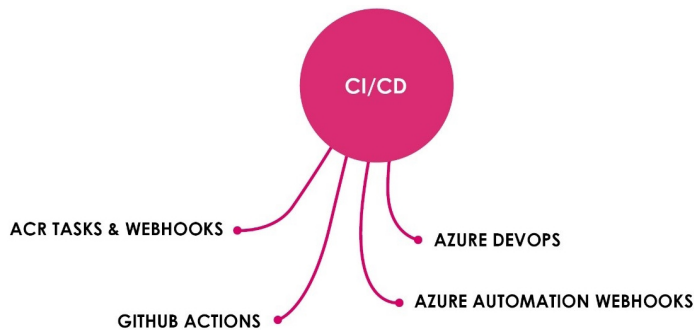


Figure 2.9 – Zoom in on CI/CD

Our CI/CD zoom-in map includes four elements:

- **ACR TASKS & WEBHOOKS**
- **GITHUB ACTIONS**
- **AZURE AUTOMATION WEBHOOKS**
- **AZURE DEVOPS**

We'll start with the first element on the left, **ACR TASKS & WEBHOOKS** (in *Figure 2.9*). When dealing with containers, **Azure Container Registry (ACR)** has some built-in triggers (webhooks) that react to source code changes and base image updates, which facilitates container-based application deployments.

Our second element is **GITHUB ACTIONS**. A current trend in the DevOps space is to abandon **Azure DevOps (ADO)** in favor of **GitHub**, a globally well-known platform (also owned by Microsoft). At this stage, ADO is more in line with typical enterprise requirements (such as auditing, security granularity, and so on), but GitHub rapidly fills the gaps in that area. GitHub also has an enterprise version in two flavors: Enterprise Server and Enterprise Cloud. GitHub Actions automates your workflows, including CI/CD, build, test, and deployment. Given the features that GitHub provides, the GitHub option should be preferred over others for companies building their CI/CD platform from scratch. Nevertheless, you should rest assured that ADO will still be around for a long time.

Our third element, **AZURE AUTOMATION WEBHOOKS**, is a feature of Azure Automation that allows an external service (such as ADO, GitHub, and so on) to start a runbook (your automation task).

CI/CD means tooling. **AZURE DEVOPS** (our fourth node) is Azure's historically preferred tool to provision Azure resources and deploy applications. It exists in multiple flavors: Azure DevOps Services and Azure DevOps Server. Its ancestors were **Team Foundation Server (TFS)** and **Visual Studio Team Services (VSTS)**. A common belief about ADO (certainly due to its name) is to think that it can only be used for Azure. This is, of course, totally wrong! ADO can be used to target any system – on-premises, AWS, and so on – but ADO has native tools to facilitate Azure deployments. ADO makes use of Microsoft-hosted agents, as well as self-hosted agents, whenever private connectivity is required.

We will explore the factory concerns more deeply in *Chapter 4, Infrastructure Deployment*, when we explain IaC.

Learning about identity

In the cloud, **identity** is one of the most important layers, and it is often overlooked or unknown (or not well-known) by solution and security architects. There is a huge difference between on-premises identity systems and cloud-based identity systems, where OpenID Connect and OAuth are dominant. *Figure 2.10* zooms in on the identity category:

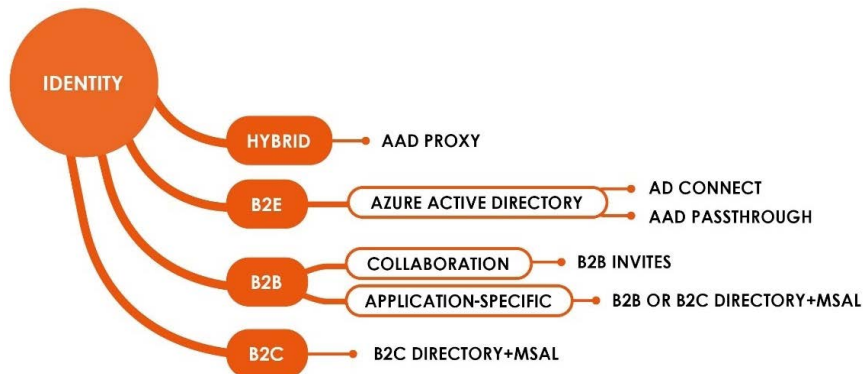


Figure 2.10 – Zoom in on identity

For the **IDENTITY** zoom-in map, we have four top-level groups:

- **HYBRID**
- **B2E**
- **B2B**
- **B2C**

In the **HYBRID** group, the AAD proxy gives you remote, secure access to your on-premises apps.

Active directory and **Azure Active Directory (AAD)** do not have that much in common when closely looking at them. AAD exposes many features for which similar concepts do not even exist on-premises. **Identity and Access Management (IAM)** teams have a solid learning curve in front of them. AAD is the cornerstone of Azure. Identity is one of the key foundations that you must define before you host anything in Azure. It is an important part of the Cloud Adoption Framework and a solid foundational asset. An Azure solution architect must know the bare minimum of modern authentication machinery. AAD addresses both infrastructure and application concerns.

That brings us to our second group in the identity zoom-in map. In a **Business-to-Enterprise (B2E)** context, AAD is used to authenticate internal employees and collaborators to systems and applications (this is also sometimes called business-to-employee.) **Azure AD Connect** provides hybrid identity features, such as password hash sync, pass-through authentication, synchronization between your on-premises directory and AAD to let you leverage single sign-on, as well as health monitoring features. Azure AD Passthrough is an alternative to **Active Directory Federation Services (ADFS)** that allows you to validate user credentials against your on-premises directory, while not having to plan for the full ADFS infrastructure. On top of it, Azure AD Passthrough can seamlessly switch to AAD authentication, should the on-premises agent not be available.

Next is B2B, our third grouping from *Figure 2.10*. AAD can also be used in a B2B context through B2B invites. That is typically how you can control who Office 365 users can invite in a B2B collaboration context. The same applies to custom workloads.

Finally, we have our fourth/bottom group. In a B2C market with many public-facing apps and APIs, Azure AD B2C is the preferred choice, since it optionally integrates with social identity providers, and it proposes valuable B2C policies targeted to Lambda consumers.

Because identity is an important security pillar in the cloud, we will explore it further in *Chapter 7, Security Architecture*.

Learning about security

Security is everywhere. *Figure 2.11* is a high-level view of Azure's security landscape:

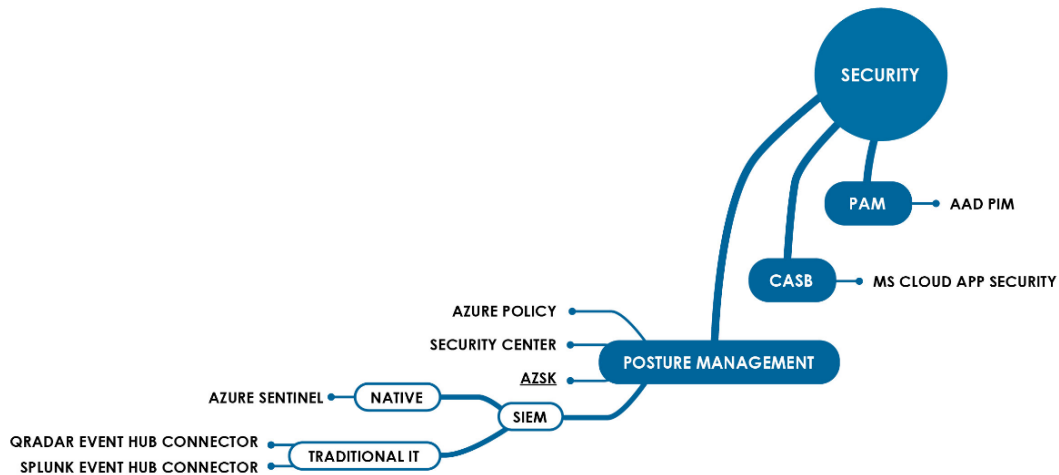


Figure 2.11 – Zoom in on security

We have three top-level groups for our **SECURITY** zoom-in map (*Figure 2.11*):

- **POSTURE MANAGEMENT**
- **CASB**
- **PAM**

Let's start with the posture management grouping. **Azure Policy** compares your resource properties to its business rules, in order to evaluate how you're using your resources. For a solution architect, it is important to know about transversal concerns, such as which SIEM systems they have to integrate with, and that a service such as **Azure Security Center** might help them detect security issues in pre-production environments (early enough in the life cycle of their solution).

The **Azure Security Kit (AzSK)** is underlined on our map because it is no longer futureproof. In the past, it used to compensate for what Security Center lacked, regarding PaaS/**Function as a service (FaaS)** service coverage. Security Center was initially only focused on IaaS components, such as virtual machines, while AzSK is mostly focused on PaaS and FaaS. Now that Security Center also covers the main PaaS services, as well as containers, it overshadows AzSK in that matter.

Azure Sentinel is Azure's integrated **Security Information and Event Management (SIEM)**. It was created by Microsoft to have a cloud-native, scalable, and integrated SIEM. Azure Sentinel can integrate with various data sources, including non-Azure ones. Advanced incident detection and response scenarios can be built through the use of AI-enabled notebooks. At the time of writing, Azure Sentinel is not yet at the level of its competitors and does not appear yet on Gartner's magic quadrant. However, Azure Sentinel (like most cloud-native services) is quickly catching up.

We will explore more deeply all the security concerns in *Chapter 7, Security Architecture*.

Learning about connectivity

Like identity, **connectivity** is a very important pillar. While not necessarily knowing bits and bytes about networking in Azure, solution architects should understand what role it plays in the overall solution. Every asset might deal with public and private endpoints, with different types of reverse proxies and firewalls. When it comes to hybrid workloads, say, for instance, a frontend in the cloud talking to an on-premises backend, connectivity becomes even more crucial. Cross-cutting concerns, such as performance and resilience, are directly subject to the underlying connectivity plumbing. *Figure 2.12* shows the most important connectivity options at our disposal, in order to bridge Azure data centers to on-premises ones, as well as to route and secure traffic:

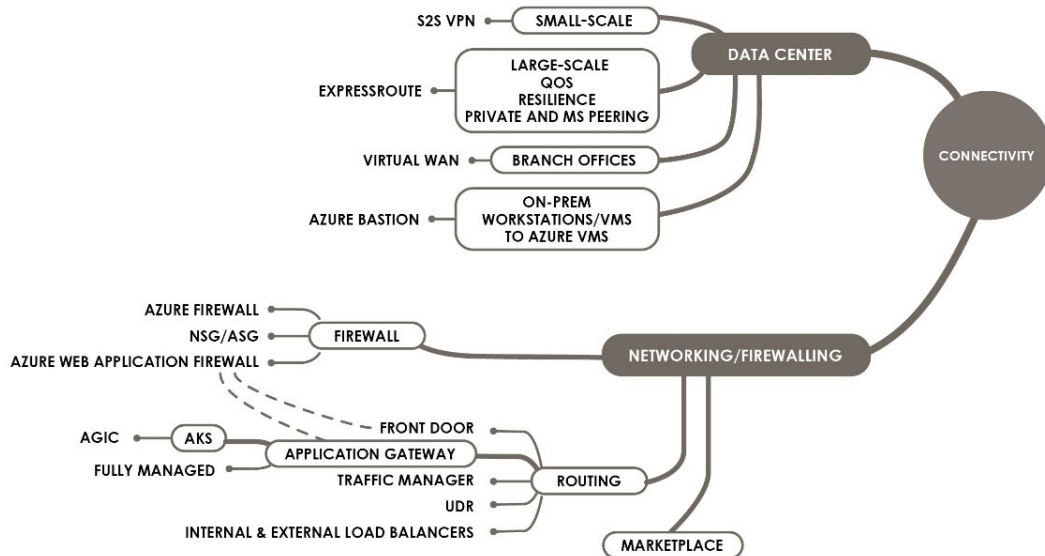


Figure 2.12 – Zoom in on connectivity

For the connectivity zoom-in map, we have two top-level groups:

- **DATA CENTER**, which you have to understand as an on-premises data center
- **NETWORKING/FIREWALLING**

Let's start with **DATA CENTER**. **Azure ExpressRoute** guarantees a certain bandwidth, a specific level of resilience, and a quality of service, but a mere VPN connection does not. Solution architects should know what is already set up (if anything), in order to evaluate whether (or not) the underlying network plumbing satisfies the non-functional requirements. Azure ExpressRoute is the de facto connectivity choice made by many organizations.

The second top-level group is networking/firewalling, which we will discuss further in *Chapter 3, Infrastructure Design*, and *Chapter 7, Security Architecture*.

Since solution architects are the guardians of end-to-end architecture, they need to explore the other maps in the book to have a better understanding of the big picture.

Learning about governance/compliance

Similarly, while not being directly in charge of the overall governance, solution architects should understand how the landing zone is governed. This plays a key role in the service model, and the level of SLA their solution may offer to customers (whether internal or external customers). Let's take a look at our zoom in on governance, shown in *Figure 2.13*:

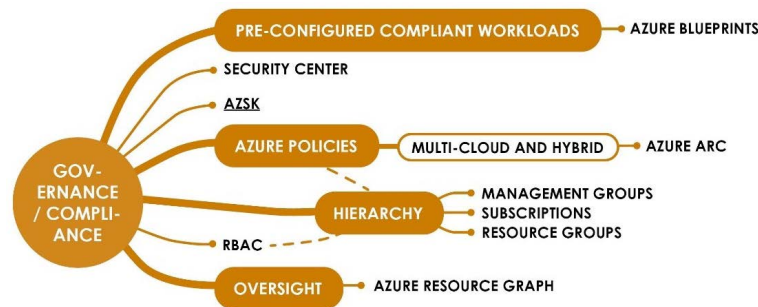


Figure 2.13 – Zoom in on governance

The **GOVERNANCE** zoom-in map has four top-level groups:

- **PRE-CONFIGURED COMPLIANT WORKLOADS**
- **AZURE POLICIES**
- **HIERARCHY**
- **OVERSIGHT**

Let's start with Azure Policy. Azure governance is directly linked to the strategy of the cloud journey, which we discussed in *Chapter 1, Getting Started as an Azure Architect*. The good news is that a documented strategy can be enforced in a tangible manner, through **Azure policies**. As a solution architect, whether designing a solution for your own organization or for a customer, you must know which policies are enforced in the hosting environment. This must be anticipated, in order to avoid any unwanted surprises later on. For instance, you might be using APIM in the Basic or Standard tier in your solution diagram. Later on, the target hosting environment might include policies that prevent the creation of public IP addresses, which will directly prevent the provisioning of your APIM instance. You should always ask for policies before starting anything. Similarly, **Role-Based Access Control (RBAC)** rules who can do what with Azure, including DevOps pipelines used to deploy solutions.

Azure Policy is tightly coupled to the hierarchy, our second top-level group. The hierarchy reflects the structure of an organization, as well as all the scopes over which Azure policies are applied. You can define a hierarchy without policies, but in practice, it never happens, except in pure sandbox environments. The hierarchy is composed of management groups, subscriptions, and resource groups. Each of these levels is an RBAC and policy scope. In practice, you might have hierarchies that are business-driven, as shown by *Figure 2.14*:

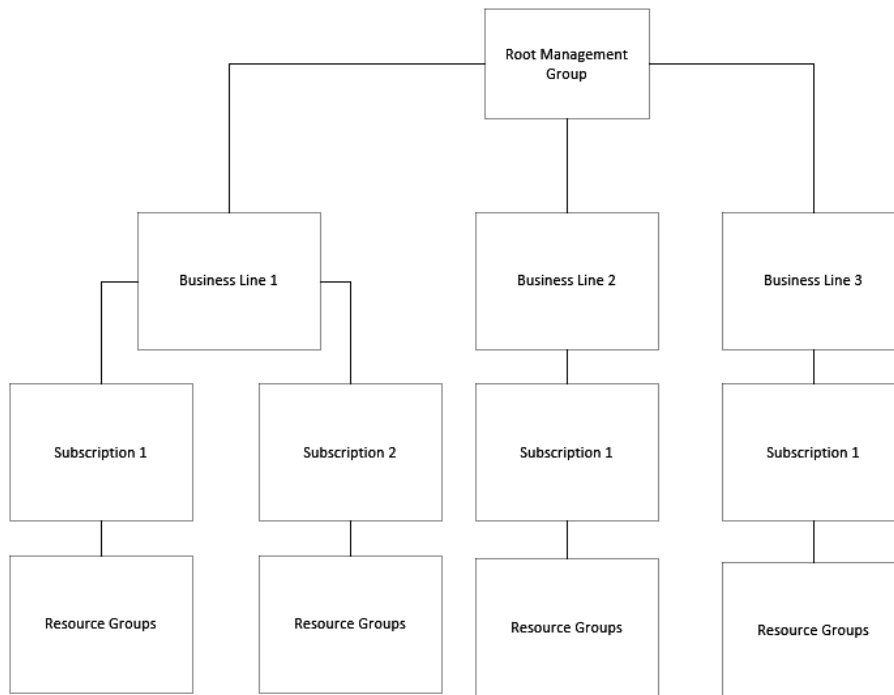


Figure 2.14 – Hierarchy with business lines

Hierarchies might also be more IT-driven, as shown in *Figure 2.15*:

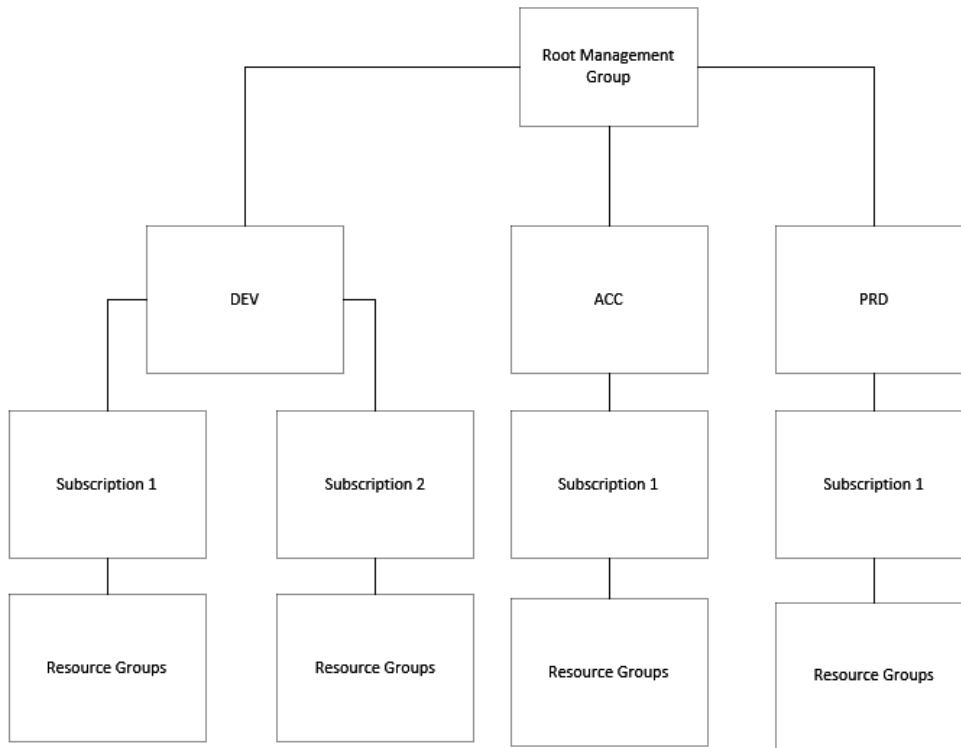


Figure 2.15 – IT-driven hierarchy

You may combine both business and IT groups by using nested management groups.

Our third top-level group is **OVERSIGHT**, which only contains Azure Resource Graph, a tool that facilitates resource exploration. It is somehow similar to a **configuration management database (CMDB)** repository.

Last but not least, pre-configured compliant workloads, our fourth top-level group, may be achieved through the use of Azure Blueprints. The purpose of blueprints is to combine IaC techniques with proper policy and RBAC assignments, so as to adhere to the organization's standards.

Admittedly, the global landing zone is often handled by infrastructure and security architects. However, the solution architect, who must think about *all* the aspects of a solution, should always gather as much information as possible about the landing zone. Even better, solution architects should try to contribute to the definition of the landing zone, so as to reflect more than only security and infrastructure aspects.

Looking at cross-cutting concerns and the cloud journey

You cannot fully address every cross-cutting concern on day one. Otherwise, day one will be postponed over and over again. In your cloud journey and strategy, you should include different maturity levels, integrate them in a roadmap, and map them to the different workload types and their associated business values and risks. Remember that you do not build your on-premises infrastructure and services in a week. This approach allows you to start small and grow over time, while delivering your solution in a timely fashion.

Now that we have browsed the different workload types and some of the typical cross-cutting concerns, let's zoom in on a very popular topic: **containerization**.

Zooming in on containerization

Containers are everywhere and on everyone's lips! In the following sections, we will explore Azure's container offering.

The Azure platform supports different flavors, which range from single-container support to full orchestrators. The solution architecture map already describes the different high-level use cases. Therefore, let's zoom deeper with a richer map that specifically targets containers (see *Figure 2.16*):

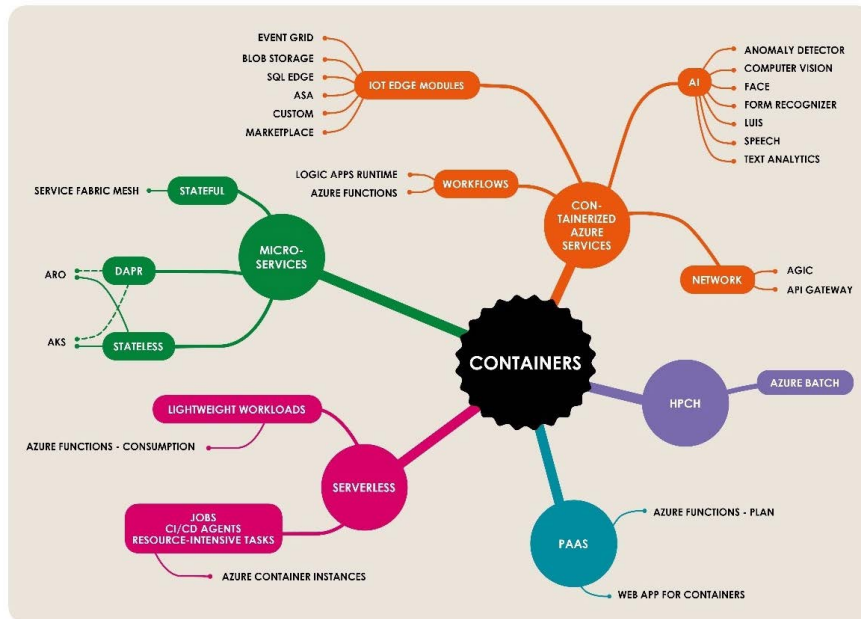


Figure 2.16 – Zoom in on containers

Microservices are one of the top use cases for running container orchestrators, such as **AKS**. **Service Fabric Mesh** has been designed at its core to deal with microservice architectures, by providing both stateless and stateful services. However, over the past 2 years, the adoption of Kubernetes worldwide has grown so fast that Microsoft's focus has now shifted to AKS. To bring statefulness (and more) to services in your AKS cluster, you can leverage **Distributed Application Runtime (Dapr)**, which brings an abstraction layer between the application code and its state stores, secret stores, and so on. Dapr also brings the actor model to Kubernetes. Dapr is multi-cloud, and it has connectors to many stores, as illustrated by *Figure 2.17*:

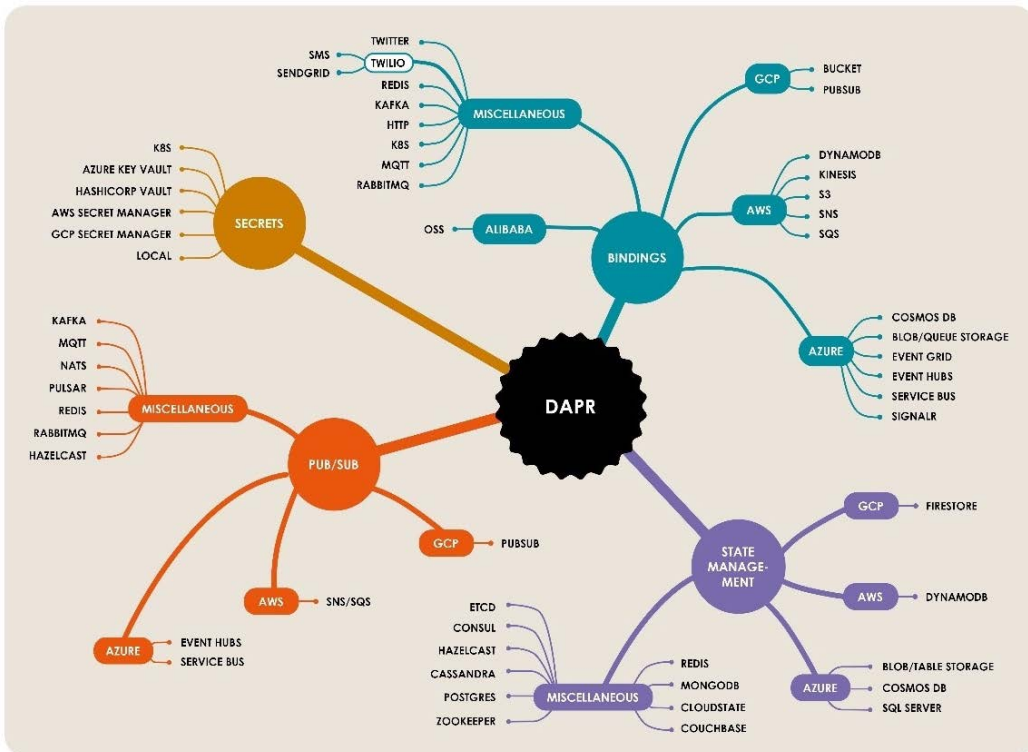


Figure 2.17 – Dapr bindings

Coming back to *Figure 2.16*, **Azure Functions** is available on both the serverless and PaaS offerings. The pricing model is not the only difference between the two flavors. PaaS-hosted Azure Functions have more capabilities, such as integrating with virtual networks. The following link provides a great view of all the network options, from the perspective of Azure Functions hosting: <https://docs.microsoft.com/azure/azure-functions/functions-networking-options>.

Many Azure services can be hosted as containers, which is useful when you have workloads running at the Edge, multi-cloud workloads, and hybrid deployments. Of course, when you self-host an Azure service, you are responsible for ensuring high availability and disaster recovery (which is an end-to-end scenario). You have to bring your own compute, but you have the advantage of being able to run the service within a perimeter that you control. It is still very important to assess the pros and cons of self-hosting a service.

Other dimensions (such as costs, complexity, and the level of operations incurred by each containerization approach) are very important too. *Figure 2.18* highlights these dimensions:

	Cost	Complexity	Operations
Azure Kubernetes Services	Medium to high	High	High
Azure Red Hat OpenShift	Medium to high	Medium	Medium
Service Fabric Mesh	Low	Medium	Low
Azure Container Instances	Low	Low	Low
Azure Batch	Medium to high	Medium	Low
Web App for Containers	Medium	Low	Low
Azure Functions on consumption	Low	Low	Low
Azure Functions on PaaS	Medium	Low	Low
Self-hosted containers versus a Microsoft-hosted counterpart	Higher	Higher	Higher

Figure 2.18 – A comparison table of container options

You should always try to follow a low-low-low path whenever applicable (such as ACI and Azure Functions on consumption), since it will be easier, cheaper, and quicker for your time-to-market.

Solution architecture use case

In the following sections, we will focus on a concrete use case (description follows). Our objective is to help you build a reference architecture, by using the map as your Azure compass to find the relevant options for a given business scenario.

Looking at a business scenario

Since we decided to zoom in a little more on containerization in this chapter, we will demonstrate one possible usage of containers in a workflow-like scenario.

For our example, we will consider the following requirements:

Contoso needs a configurable workflow tool that allows you to orchestrate multiple resource-intensive tasks. Each task must launch large datasets to perform in-memory calculations. For some reason, the datasets cannot be chunked into smaller pieces, which means that memory contention could quickly become an issue under a high load. A single task may take between a few minutes to an hour to complete. Workflows are completely unattended (no human interaction) and asynchronous. The business needs a way to check the workflow status and be notified upon completion. Also, the solution must be portable. Contoso's main technology stack is .NET Core. Of course, this should have been done yesterday, and there is not much budget allocated to the project.

These few constraints might sound very familiar to you! The next section focuses on the most important keywords that should draw your attention.

Using keywords

As an Azure solution architect, you must capture the essential part of a story. Here are a few keywords that can structure your train of thought when building the solution. Let's review them one by one:

- **Portability:** Whenever portability comes as a requirement, containers should be the default answer.
- **.NET Core:** The company often wants the project to be realized as soon as possible, with the technology stack they already master. We cannot bring in a solution that would be too disruptive skills-wise.
- **Resource-intensive tasks:** In the cloud, like in every environment, there is no free lunch. If you want compute power, you must pay for it. Given that we have a low budget, we won't be able to afford plain virtual machines with high memory and CPU profiles.

There is, however, a way to reduce costs by only paying for the allocated resources when you need them. We'll expand on the concept of a **serverless** approach, which we mentioned in the first chapter.

- **Task duration:** This criterion is a structuring factor, as it eliminates some hosting options, such as Azure Functions hosted on the consumption pricing tier (which cannot exceed 10 minutes of execution). One option could be Azure Functions on pre-paid pricing tiers.
- **Workflow:** A workflow is a sequence of steps that are executed in a coordinated way. This aspect is important because it reduces the field of possibilities.

Let's now see how to make use of the map to progress in our thinking process.

Using the solution architecture map against the requirements

Now that we have highlighted the important keywords, let's take a look at our map to try and make sense of it. We'll form a reference architecture, which you could reuse for other projects later (where you have similar needs). We will now look at some workflow/orchestration capabilities, which is one of our map concerns. *Figure 2.19* is a subset of the solution architecture map:

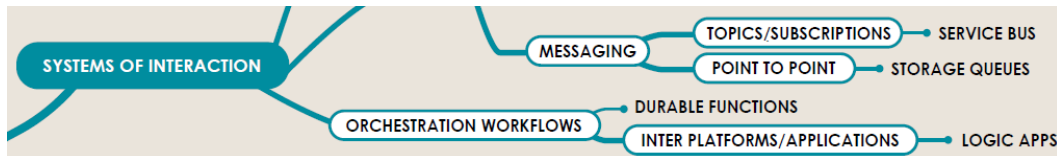


Figure 2.19 – The workflow capabilities

A workflow is an orchestration, and we see two possibilities (under **ORCHESTRATION WORKFLOWS** in *Figure 2.19*): **DURABLE FUNCTIONS** and **LOGIC APPS**. Logic Apps seems more appropriate for integration scenarios. Since our workflow is scoped to a single application, Durable Functions might be a fit. As you can see, it is hard to make a choice that is based on *Figure 2.19*. *Figure 2.20* is another small map that may help you choose between Logic Apps and Durable Functions, beyond the scope of our scenario:

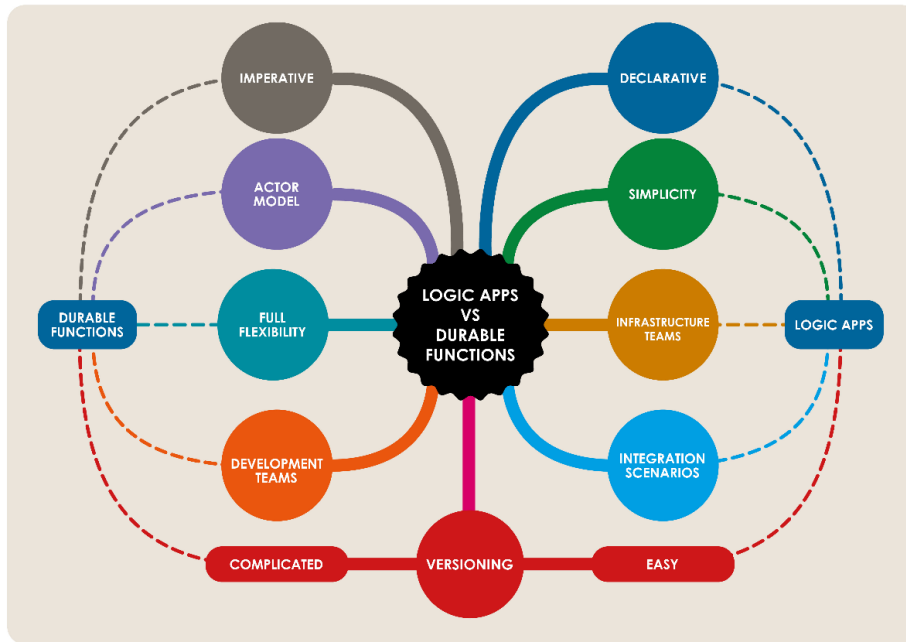


Figure 2.20 – A map focused on Logic Apps and Durable functions

The manpower you have at your disposal is one of the factors to consider. Logic Apps is fully declarative, and it does not require any programming skills. Durable Functions is mostly developed for the scope of a single application. We will therefore consider using Durable Functions for our scenario, because we know we have .NET developers, and we do not have to deal with a workflow that goes beyond the scope of our single solution. Beyond this book, the map will help you find relevant services for a given use case, but it is up to you to dig further and understand exactly what is behind the service in question. Since we are in the context of the book, we can cheat a bit and give you an extra explanation about Durable Functions, which we have just selected. As stated before, it is particularly useful for workflow-like workloads. It differs from mere Azure Functions in that it is stateful and not stateless. A workflow needs to persist its state and resist a temporary outage, in order to resume to the correct point in time when the outage is over. Azure Durable Functions leverages the durable framework, which persists the state into a storage account for you. It will automatically watch for events that are related to the orchestration and it will ensure a proper follow-up of the different activities. We will see the durable framework in action later in this use case.

Similarly, *Figure 2.21* (a subset of the Azure solution architecture map) shows cost-friendly and potentially resource-intensive-friendly services, under the containerization concern:

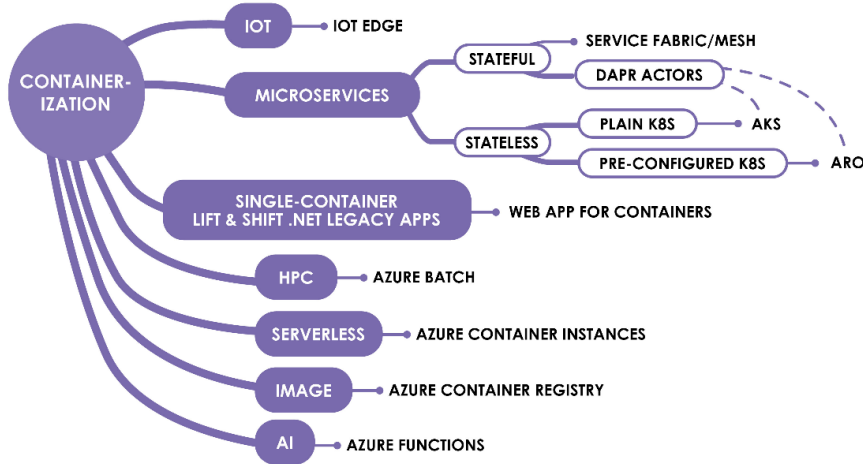


Figure 2.21 – The container options

Looking at our use case, it might be a good fit to go with the serverless nature of ACI as a hosting platform to host and execute our workflow tasks (especially for our limited budgets). The map alone is not sufficient to decide between ACI and Azure Batch, in order to fit with our resource-intensive requirements. As an Azure solution architect, you need to further analyze the difference between ACIs and Azure Batch. This is an example that shows the limits of the maps. They will help you get the big picture, but you have to do your homework (with further analysis and research) at some point.

After your analysis, you realize that ACI may also be used for resource-intensive tasks, and it is capable of running both Windows and Linux containers, which is fully in line with our technology stack. In the next section, we will see how to infer a reference architecture from these preliminary conclusions.

Building the target reference architecture

With this progress, you are ready to build your reference architecture. Of course, there is never a one-size-fits-all approach, but *Figure 2.22* shows you a possible solution:

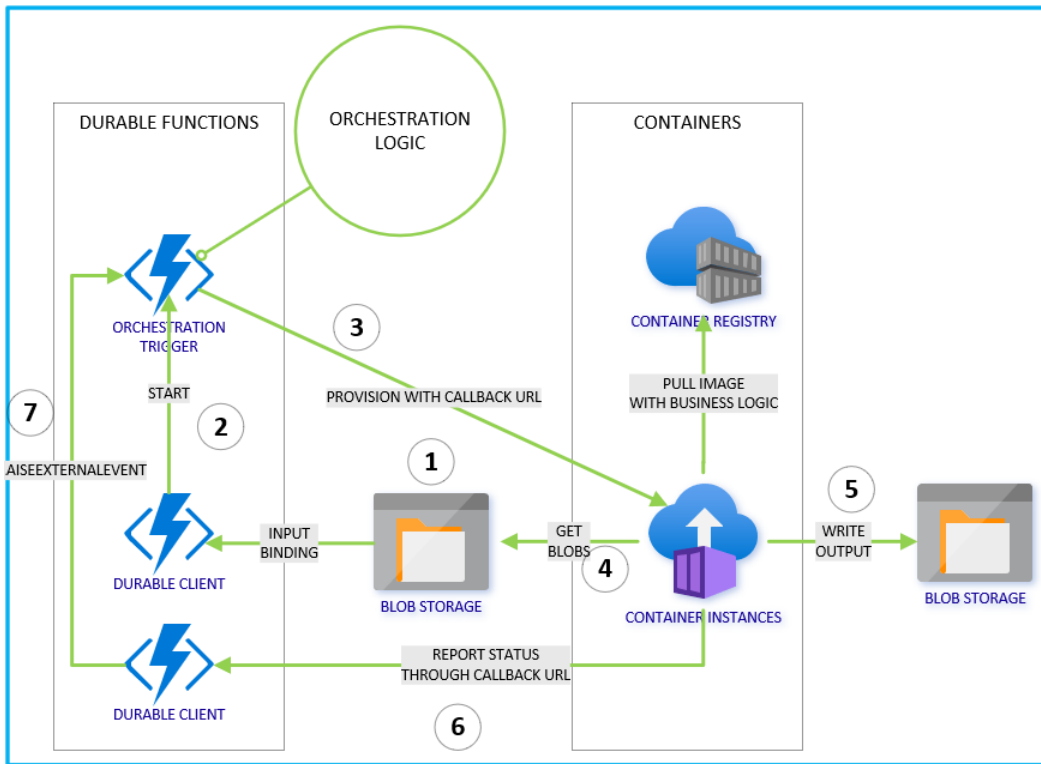


Figure 2.22 – A sample reference architecture

There are small numbers on the diagram that we explain as follows:

1. **Azure Blob storage**, part of a storage account, receives the incoming blobs to be treated by the workflow. These are the large files we referred to in our scenario.
2. Upon the setup of Blob storage, a durable client (that has a Blob storage binding) is kicked off. Later, it starts a new orchestration, while passing some orchestration configuration (such as the number of steps, retries, timeouts, and so on).
3. The main orchestrator, which could have sub-orchestrators, in turn provisions one container instance per workflow step, and it passes a callback URL that is used by ACI to report its status and to optionally return a pointer to the input data of the next task.
4. ACI gets the input blob that it needs to handle.
5. ACI writes output to another Blob storage.

The overall process contains one or more steps, and each step is allocated a dedicated ACI with the required compute, which is a maximum of four CPUs and 16 GB of RAM (at the time of writing). Each task may execute in parallel or one after the other, depending on the orchestrator logic. *Figure 2.23* is an iteration of a sequential workflow that could be handled by our reference architecture:

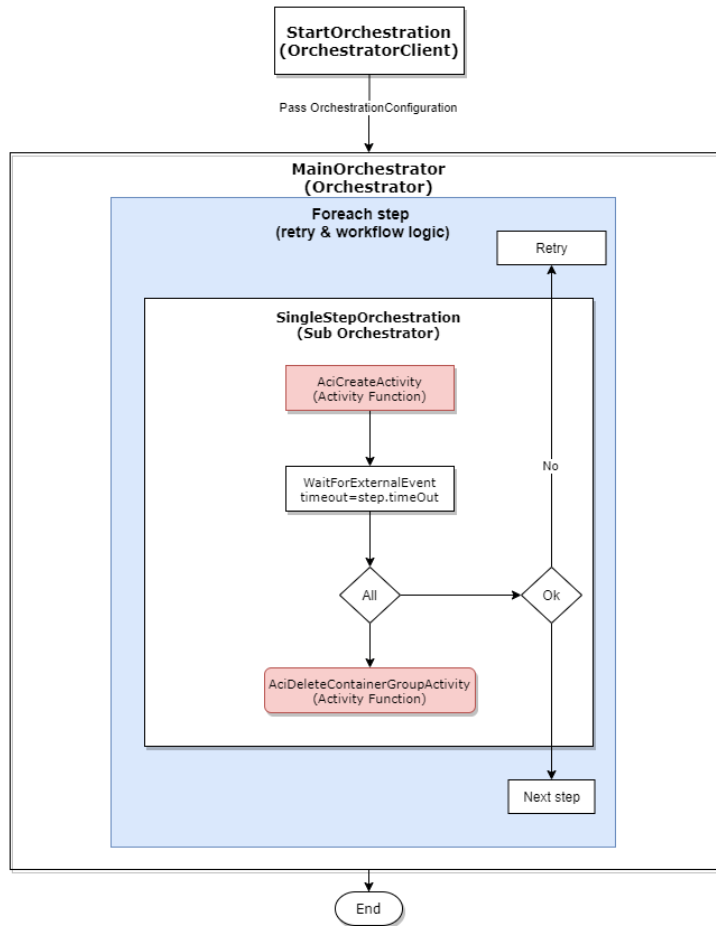


Figure 2.23 – A sequential workflow example with Durable Functions and ACI

The main orchestrator starts with one sub-orchestrator per workflow task. Each task creates a container instance and waits for its feedback through the **WaitForExternalEvent** operation. The Durable Functions framework allows you to handle retries and timeouts. Whatever the workflow step status is (such as failed or timed out), the corresponding ACI is deleted, and the main orchestrator either retries, stops the orchestration, or goes to the next step. Such orchestration logic could be injected dynamically by the orchestrator client.

The following code snippet is an example of a configuration workload that we can pass to a new orchestration instance:

Important note

The full .NET project code, which includes all the following code snippets, is available on the GitHub repository here: <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/tree/master/Chapter02/code/PacktOrchestrationDemo>:

```
{
  "isRewindable": true,
  "defaultStepTimeout": 5,
  "steps": [
    {
      "stepName": "step1",
      "maxRetryCount": 2,
      "stopFlowOnFailure": true,
      "timeout": 5
    },
    {
      "stepName": "step2",
      "maxRetryCount": 0,
      "stopFlowOnFailure": false,
      "timeout": 3
    }
  ]
}
```

We instruct the orchestrator to run two steps and that the workflow is stopped if step 1 fails more than three times. Let's explore this solution further by looking at some code.

Code view of our workflow-based reference architecture

Next, let's explore a code sample in .NET Core that implements this architecture. Note that this is only for demonstration purposes and is not intended to be used directly in production.

As shown in the previous section, we can inject the payload via a JSON HTTP POST body. The following two classes are the object presentation of this JSON payload:

```
public class OrchestrationConfiguration{
    public bool isRewindable { get; set; }
    public int defaultStepTimeout { get; set; }
    public List<OrchestrationStep> steps { get; set; }
}

public class OrchestrationStep{
    internal string instanceId { get; set; }
    public string stepName { get; set; }
    public int maxRetryCount { get; set; }
    public bool stopFlowOnFailure { get; set; }
    public int timeOut { get; set; }
}
```

The next Azure Durable Functions can be triggered like regular functions by using proper bindings. In this example, we use an HTTP trigger binding, which starts a new orchestration:

```
[FunctionName("StartOrchestration")]
public static async Task<HttpResponseMessage>
    StartOrchestration(
        [HttpTrigger(AuthorizationLevel.Anonymous, "post")]
        HttpRequestMessage req,
        [DurableClient] IDurableOrchestrationClient starter,
        ILogger log){

    string instanceId = await starter.StartNewAsync(
        "MainOrchestrator",
        JsonConvert.DeserializeObject<
            OrchestrationConfiguration>(
```

```

        await req.Content.ReadAsStringAsync());
    return starter.CreateCheckStatusResponse(
        req, instanceId);
}

```

In the real world, you would place this function behind an API gateway. The `IDurableOrchestrationClient` output type includes a function that interacts with an orchestrator, to either start a new one or to provide some feedback to a running orchestration. In the preceding piece of code, we start a new orchestration and pass the deserialized JSON body.

Next, we run the orchestration itself:

```

[FunctionName("MainOrchestrator")]
public static async Task MainOrchestrator(
    [OrchestrationTrigger] IdurableOrchestrationContext
        context, ILogger log)
{
    OrchestrationConfiguration process =
        context.GetInput<OrchestrationConfiguration>();

    foreach (OrchestrationStep step in process.steps)
    {
        var retryOptions = new RetryOptions(
            firstRetryInterval: TimeSpan.FromSeconds(5),
            maxNumberOfAttempts: (step.maxRetryCount > 0) ?
                step.maxRetryCount + 1 : 1);

        try
        {
            step.timeOut = (step.timeOut == 0) ?
                process.defaultStepTimeOut : step.timeOut;
            var result = await context
                .CallSubOrchestratorWithRetryAsync<bool>(
                    nameof(SingleStepOrchestration),
                    retryOptions, step);
        }
    }
}

```

```
        catch
        {
            if (!context.IsReplaying)
            {
                log.LogWarning("step {0} failed",
                    step.stepName);
                if (step.stopFlowOnFailure)
                    break;
            }
        }
    }
    log.LogInformation("ORCHESTRATION IS OVER");
}
```

For each step injected by our configuration payload, we start a sub-orchestration, with the retry options and timeout defined. We check whether the workflow should stop, in case the current step failed. In our example, we injected two steps in the payload. The first step has its `stopFlowOnFailure` property set to `true`, so if step 1 fails, the workflow stops.

Next, we have the step-related code itself:

```
[FunctionName(nameof(SingleStepOrchestration))]
public static async Task<bool> SingleStepOrchestration(
    [OrchestrationTrigger] IdurableOrchestrationContext
        context,
    ILogger log)
{
    try
    {
        var step = context.GetInput<OrchestrationStep>();

        log.LogInformation("CREATING ACI for {0} on instance
            {1} ",
            step.stepName, context.InstanceId);
        log.LogInformation("CALLBACK URL
            http://localhost:7071/api/stepResultCallback?
            instanceId={0}&eventName={1}&status= ",
            context.InstanceId, step.stepName);
        log.LogInformation("WAITING FOR EVENT {0} on
```

```
        instance {1}", step.stepName,
        context.InstanceId);

    var state = await
        context.WaitForExternalEvent<string>(
            step.stepName, TimeSpan.FromMinutes(step.timeOut));

    log.LogInformation(
        "STEP {0} INSTANCE IS {1} STATE IS {2}",
        step.stepName, context.InstanceId, state);

    if (state != "ok")
    {
        log.LogInformation("state NOK");
        throw new ApplicationException();
    }

}

catch(TimeoutException)
{
    log.LogInformation("TIMEOUT");
    throw;
}

finally
{
    log.LogInformation("deleting ACI");
}

return true;
}
}
```

For the sake of brevity and simplicity, we do not really provision a container instance. We simply log the fact that we would do so. We generate a callback URL to pass to ACI and we let it report its status to the orchestrator.

Next, we wait for an external event, which is nothing more than the ACI feedback, which we will simulate in the next section. The timeout makes sure we do not wait forever, should the ACI crash unexpectedly or remain in a hanging state. Once we receive the feedback, we check whether it is okay or not (admittedly in a naïve way here) to report it back to the main orchestrator. In the case of an issue, we throw an exception, in order to make sure the main orchestrator catches it.

Here is the function that ACI calls back (thanks to the provided callback URL) in order to report its status:

```
[FunctionName("stepResultCallback")]
public static async Task stepResultCallback(
    [HttpTrigger(AuthorizationLevel.Anonymous, "post")]
    HttpRequestMessage req,
    [DurableClient] IDurableOrchestrationClient cli,
    ILogger log)
{
    string eventName = req.RequestUri.ParseQueryString()
        .Get("eventName");
    string status = req.RequestUri.ParseQueryString()
        .Get("status");
    log.LogInformation($"RECEIVED FEEDBACK FROM {eventName}
        WITH STATUS {status}");
    await cli.RaiseEventAsync(req.RequestUri.
        ParseQueryString().Get("instanceId"),
        eventName, status);
}
```

We basically raise the event back to the sub-orchestrator by using its instance identifier. In the preceding example, we again use an HTTP trigger, but we could bind this to a queue where ACI would drop messages to return its feedback. Here, again, the function would be placed behind an API gateway and would be secured using either a function key or an AAD access token.

For your information, one of the Durable Functions gotchas is *versioning*, because you need to make sure that you do not inject breaking changes. You must also work with deterministic APIs only. Going deeper into this is a prerogative of the application architect or a senior developer.

Let's see this code in action.

Looking at the code in action

Looking at code blocks is sometimes a little too abstract. Let's see this code in action:

1. Using Visual Studio and Fiddler, we will kick off a new orchestration. Clone the GitHub repo (<https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook>) locally.
2. Open the solution file in `/Chapter02/code/PacktOrchestrationDemo.sln` and run the project from Visual Studio 2019. *Figure 2.24* shows the Azure Functions runtime running locally:

```
Azure Functions Core Tools (3.0.2912 Commit hash: bfcbbe48ed6fdacdf9b309261ecc8093df3b83f2)
Function Runtime Version: 3.0.14287.0

Functions:

    StartOrchestration: [POST] http://localhost:7071/api/StartOrchestration
    stepResultCallback: [POST] http://localhost:7071/api/stepResultCallback
    InitializeWithBreakingChange: activityTrigger
    MainOrchestrator: orchestrationTrigger
    SingleStepOrchestration: orchestrationTrigger

For detailed output, run func with --verbose flag.
Hosting environment: Production
Content root path: C:\Users\steph\Source\Repos\packt\PacktOrchestrationDemo\bin\Debug\netcoreapp3.1
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.
Host lock lease acquired by instance ID '000000000000000000000000921E07D7'.
```

Figure 2.24 – The Azure Functions runtime running locally

Important note

If you are prompted by Windows Firewall (or another prompt), allow the action.

The program is now waiting for a new orchestration to start. We will do so by using Fiddler (you could use Postman as well). See *Figure 2.25*:

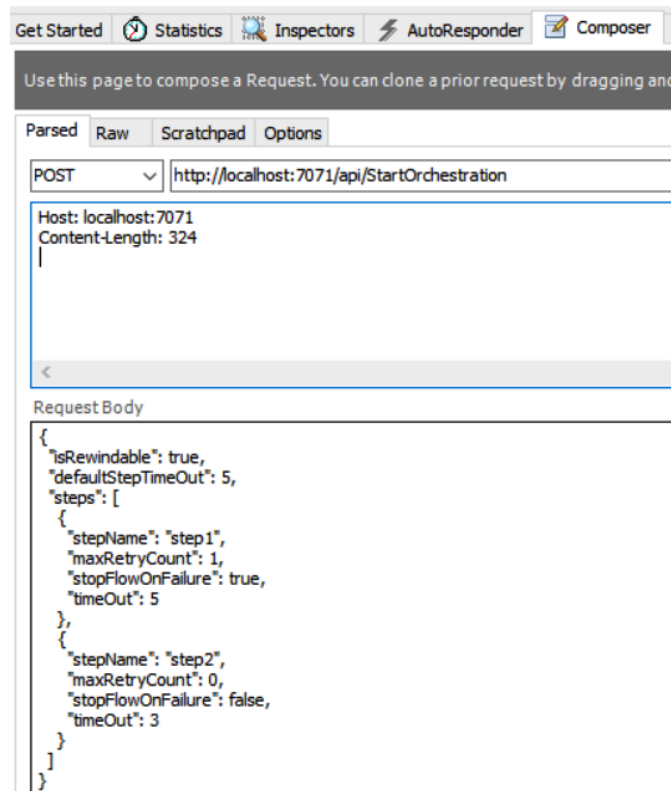


Figure 2.25 – Starting the orchestration

- Using the **Composer** tab, we craft our POST query to the starter function endpoint. We pass a sample payload, which you can download from: <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/blob/master/Chapter02/code/PacktOrchestrationDemo/samplepayload.json>, which basically states the following:

There are two steps in our workflow.

The first step is allowed to fail or time out twice (first time +1 retry).

If the first step fails twice in a row, the whole workflow stops.

Whether the second step (and the last step, in this case) fails or succeeds, this should not be retried, nor should it stop the workflow.

Upon executing the preceding Fiddler request, the orchestrator starts (see *Figure 2.26*):

```
Azure Functions Core Tools (3.0.2912 Commit hash: bfcbbe48ed6fdacdf9b309261ecc8093df3b83f2)
Function Runtime Version: 3.0.14287.0

Functions:

    StartOrchestration: [POST] http://localhost:7071/api/StartOrchestration
    stepResultCallback: [POST] http://localhost:7071/api/stepResultCallback
    MainOrchestrator: orchestrationTrigger
    SingleStepOrchestration: orchestrationTrigger

For detailed output, run func with --verbose flag.
Hosting environment: Production
Content root path: C:\Users\steph\Source\Repos\packt\PacktOrchestrationDemo\bin\Debug\netcoreapp3.1
Now listening on: http://0.0.0.0:7071
Application started. Press Ctrl+C to shut down.
Host lock lease acquired by instance ID '00000000000000000000000921E07D7'.
-----CREATING ACI for step1 on instance e83c22e99c814e86b30ef83540548003:0 -----
-----CALLBACK URL http://localhost:7071/api/stepResultCallback?instanceId=e83c22e99c814e86b30ef83540548003:0&eventName=step1&status=
-----WAITING FOR EVENT step1 on instance e83c22e99c814e86b30ef83540548003:0-----
```

Figure 2.26 - The orchestration kicked off

- The orchestration indicates that it waits for step 1 to complete. Using the provided callback URL, we can simulate the response of ACI with Fiddler again (see *Figure 2.27*):

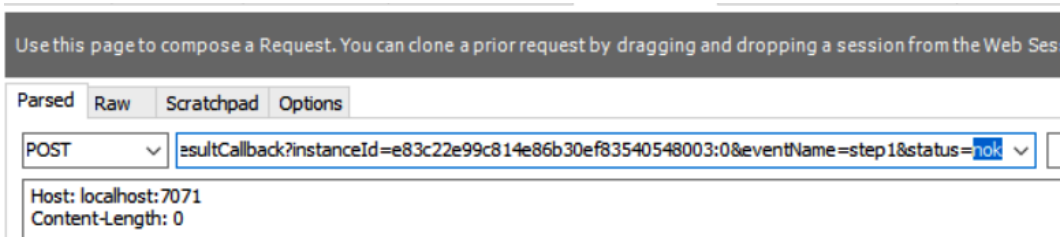


Figure 2.27 – Simulating step 1, the ACI response

In this case, we returned NOK, meaning that step 1 failed (this stands for *not okay*), and this is captured by our orchestration, as shown in *Figure 2.28*:

```
RECEIVED FEEDBACK FROM step1 WITH STATUS nok
-----CREATING ACI for step1 on instance e83c22e99c814e86b30ef83540548003:0 -----
-----CALLBACK URL http://localhost:7071/api/stepResultCallback?instanceId=e83c22e99c814e86b30ef83540548003:0&eventName=step1&status=
-----WAITING FOR EVENT step1 on instance e83c22e99c814e86b30ef83540548003:0 -----
-----STEP step1 INSTANCE IS e83c22e99c814e86b30ef83540548003:0 STATE IS nok-----
-----state NOK-----
-----deleting ACI-----
e83c22e99c814e86b30ef83540548003:0: Function 'SingleStepOrchestration (Orchestrator)' failed with an error. Reason: System.ApplicationException: Error in the application
   at PacktOrchestrationDemo.Orchestration.SingleStepOrchestration(IDurableOrchestrationContext context, ILogger log) in C:\Users\steph\Source\Repos\packt\PacktOrchestra
tion.cs:line 94
   at Microsoft.Azure.WebJobs.Host.Executors.FunctionInvoker`2.InvokeAsync(Object instance, Object[] arguments) in C:\projects\azure-webjobs-sdk-rqm4t\src\Microsoft.Azur
e.Executors.FunctionInvoker.cs:line 52
   at Microsoft.Azure.WebJobs.Extensions.DurableTask.TaskOrchestrationShim.InvokeUserCodeAndHandleResults(RegisteredFunctionInfo orchestratorInfo, OrchestrationContext c
ontext) in C:\projects\azure-functions-durable-extension\src\WebJobs.Extensions.DurableTask\Listener\TaskOrchestrationShim.cs:line 158. IsReplay: False. State: Failed. HubName: azur
e
Name: SlotName: ExtensionVersion: 2.2.2. SequenceNumber: 12.
Executed 'SingleStepOrchestration' (Failed, Id=e8297393-7e92-4169-a023-df9aab23d1fb, Duration=389ms)
System.Private.CoreLib: Exception while executing function: SingleStepOrchestration. System.Private.CoreLib: Orchestrator function 'SingleStepOrchestration' failed: Err
or
-----CREATING ACI for step1 on instance e83c22e99c814e86b30ef83540548003:2 -----
-----CALLBACK URL http://localhost:7071/api/stepResultCallback?instanceId=e83c22e99c814e86b30ef83540548003:2&eventName=step1&status=
-----WAITING FOR EVENT step1 on instance e83c22e99c814e86b30ef83540548003:2-----
```

Figure 2.28 – The failed step 1 captured

- The error is displayed, and a new callback URL is returned to retry step 1, because we said it could be retried. See *Figure 2.29*:

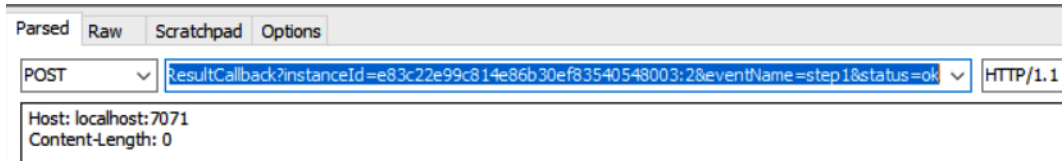


Figure 2.29 – Step 1 is retried and a status of "ok" is returned

This time, we returned a status of `ok` for step 1. This is also captured, and the orchestrator now waits for step 2 to complete, with a new callback URL. See *Figure 2.30*:

```
-----WAITING FOR EVENT step1 on instance e83c22e99c814e86b30ef83540548003:2-----
RECEIVED FEEDBACK FROM step1 WITH STATUS ok
-----CREATING ACI for step1 on instance e83c22e99c814e86b30ef83540548003:2 -----
-----CALLBACK URL http://localhost:7071/api/stepResultCallback?instanceId=e83c22e99c814e86b30ef83540548003:2&eventName=step1&status=ok-----
-----WAITING FOR EVENT step1 on instance e83c22e99c814e86b30ef83540548003:2-----
-----STEP step1 INSTANCE IS e83c22e99c814e86b30ef83540548003:2 STATE IS ok-----
-----deleting ACI-----
-----CREATING ACI for step2 on instance e83c22e99c814e86b30ef83540548003:3 -----
-----CALLBACK URL http://localhost:7071/api/stepResultCallback?instanceId=e83c22e99c814e86b30ef83540548003:3&eventName=step2&status=ok-----
-----WAITING FOR EVENT step2 on instance e83c22e99c814e86b30ef83540548003:3-----
```

Figure 2.30 – Waiting for step 2

- Now, we can return step 2's feedback. See *Figure 2.31*:

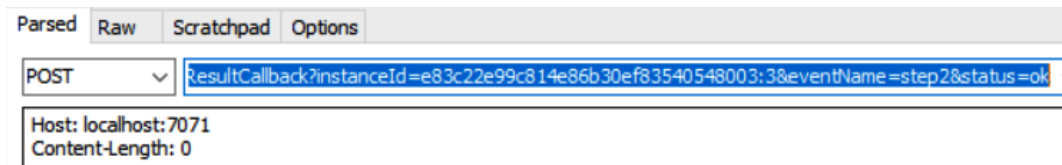


Figure 2.31 – Returning "ok" feedback for step 2

Our orchestrator captures the `ok` feedback, and it ends the orchestration, because the last step is executed. See *Figure 2.32*:

```
-----RECEIVED FEEDBACK FROM step2 WITH STATUS ok-----
-----CREATING ACI for step2 on instance e83c22e99c814e86b30ef83540548003:3 -----
-----CALLBACK URL http://localhost:7071/api/stepResultCallback?instanceId=e83c22e99c814e86b30ef83540548003:3&eventName=step2&status=ok-----
-----WAITING FOR EVENT step2 on instance e83c22e99c814e86b30ef83540548003:3-----
-----STEP step2 INSTANCE IS e83c22e99c814e86b30ef83540548003:3 STATE IS ok-----
-----deleting ACI-----
-----ORCHESTRATION IS OVER-----
```

Figure 2.32 – Orchestration is over

- You can fiddle with the configuration as you wish, in order to test the different variants, which we will not do here (as it's beyond the scope of our goal of architecture). Instead, we will show the effect of the timeout by starting a new orchestration, as shown in *Figure 2.33*:

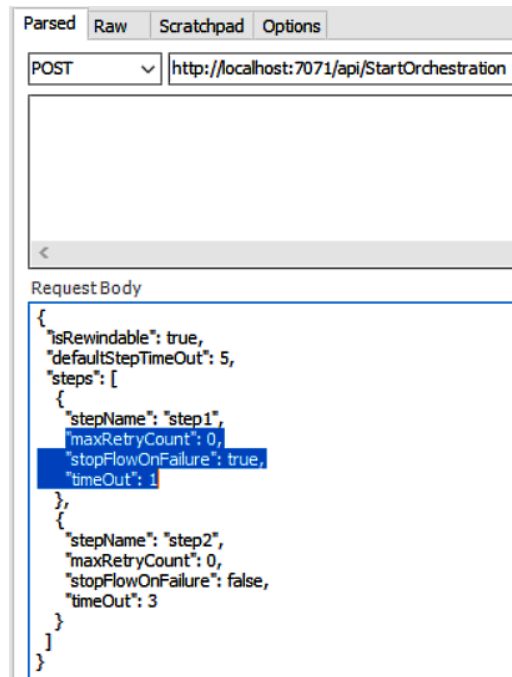


Figure 2.33 – Testing a timeout

Here, we specify a 1-minute timeout (no retry) and that the workflow should stop upon failure. A minute later, doing nothing, the orchestrator catches the timeout exception and stops the orchestration, as illustrated in *Figure 2.34*:

```
-----CREATING ACI for step1 on instance 6e2234d9a76a4c8faf07b6634fc62bd1:2-----
-----CALLBACK URL http://localhost:7071/api/stepResultCallback?instanceId=6e2234d9a76a4c8faf07b6634fc62bd1:2&eventName=step1&status-----
-----WAITING FOR EVENT step1 on instance 6e2234d9a76a4c8faf07b6634fc62bd1:2-----
-----TIMEOUT-----
-----deleting ACI-----
6e2234d9a76a4c8faf07b6634fc62bd1:2: Function 'SingleStepOrchestration (Orchestrator)' failed with an error. Reason: System.TimeoutException: Event step1 not received
at PacktOrchestrationDemo.Orchestration.SingleStepOrchestration(IDurableOrchestrationContext context, ILogger log) in C:\Users\steph\Source\Repos\packt\PacktOrche
stration.cs:line 81
at Microsoft.Azure.WebJobs.Host.Executors.FunctionInvoker`2.InvokeAsync(Object instance, Object[] arguments) in C:\projects\azure-webjobs-sdk-rqm4t\src\Microsoft.
Executors\FunctionInvoker.cs:line 52
at Microsoft.Azure.WebJobs.Extensions.DurableTask.TaskOrchestrationShim.InvokeUserCodeAndHandleResults(RegisteredFunctionInfo orchestratorInfo, OrchestrationConte
d:\a\r1\azure-functions-durable-extension\src\WebJobs.Extensions.DurableTask\Listener\TaskOrchestrationShim.cs:line 150. IsReplay: False. State: Failed. HubName:
ame: . SlotName: . ExtensionVersion: 2.2.2. SequenceNumber: 60.
executed 'SingleStepOrchestration' (Failed, Id=2621191a-0a48-4527-9868-18d604a18d0d, Duration=716ms)
System.Private.CoreLib: Exception while executing function: SingleStepOrchestration. System.Private.CoreLib: Orchestrator function 'SingleStepOrchestration' failed:
ived in 00:01:00.
-----step step1 failed-----
-----step step1 failed-----
-----ORCHESTRATION IS OVER-----
-----ORCHESTRATION IS OVER-----
```

Figure 2.34 – The timeout exception captured by the orchestrator

Note that these types of workflows, with the retry mechanisms, assume that the steps are idempotent (unchanged).

We designed our architecture (the diagram), and we even prepared some boilerplate code to create a **proof of concept**.

Let's now see, in the next section, what is still missing.

Understanding the gaps in our reference architecture

We may think we did a great job earlier when designing our reference architecture, but it suffers from important gaps. Look again at *Figure 2.35*:

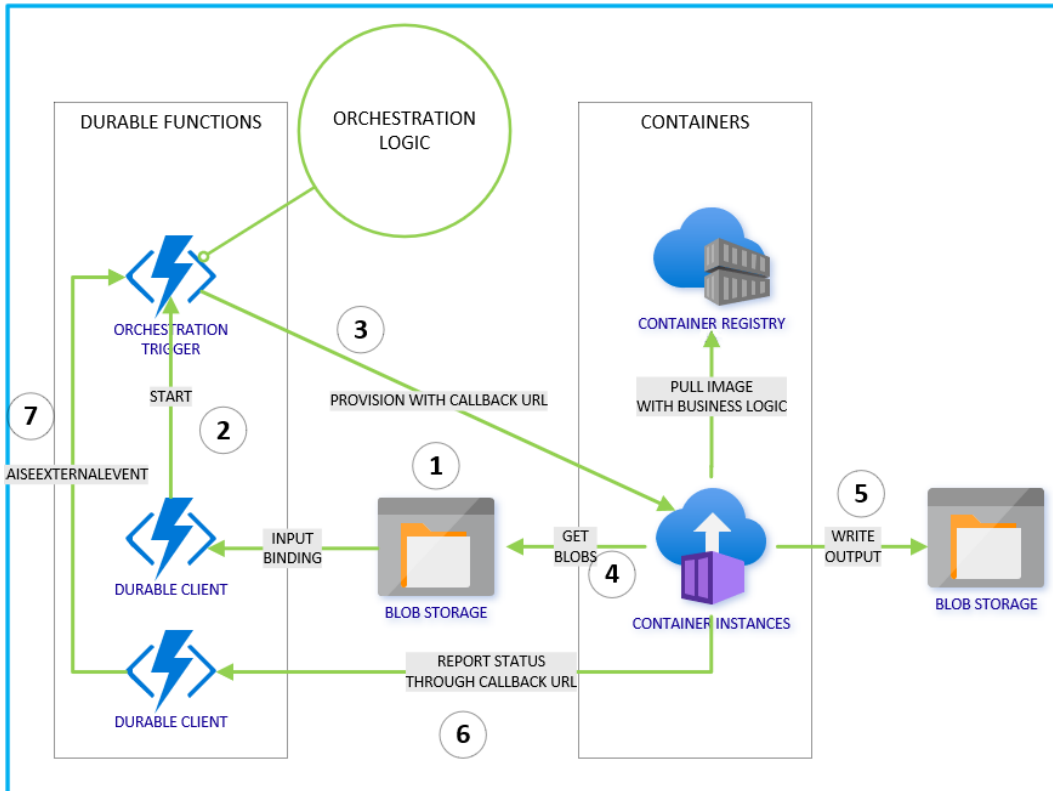


Figure 2.35 – Where are the non-functional requirements?

We have not covered the cross-cutting concerns. We mostly focused on the building blocks and their interactions, but we did not cover anything about monitoring, security, resilience, and so on. Sometimes, it can be challenging to reflect everything in a single diagram, because it makes that diagram either too big or too complex to understand. To overcome this, a possibility is to work with different views, scoped on specific areas. Doing so will also help you engage with your peers, as well as with more specialized architects.

The solution architecture map cannot go too deep into each domain, because it would simply be too broad. You can't cover a bunch of **non-functional requirements (NFRs)**. Using only this map to cover all the NFRs is not possible. As a solution architect, you will also need to refer to other maps in this book to find your way in other fields and you'll perhaps also need to ask for extra help from infrastructure and security architects. The views that should be added to this architecture are as follows:

- **Monitoring view:** This is where you add Azure Monitor, Log Analytics, and dashboards into the mix. You might have to add Splunk or any other on-premises tool that your organization (or your customer) is using. This is doable by only using the solution architecture map.
- **Security view:** This is where you indicate the different authentication mechanisms, such as **Shared Access Signature (SAS)** tokens, managed identities, OAuth flows, and so on. You also specify which API gateway policies would enter into play, as well as the secret stores, encryption mechanisms, and so on. The security map in *Chapter 7, Security Architecture*, will help you here.
- **High-availability and disaster recovery views:** Here, you focus on the availability and resilience of your solution. The infrastructure map of this chapter will help you here.
- **Deployment view:** Here, you focus on the factory and the post-deployment provisioned components.

Once you have read the other chapters and digested them, we will review this architecture with all these aspects covered! As we stated previously, we will start with the easy things, but the complexity will grow as we go.

Summary

In this chapter, we described the solution architecture map and its different classification categories, which are SoE, SoR, SoI, and systems of interaction (IPaaS). This categorization, commonly used in architecture, makes it easier to dispatch services. We provided explanations and extra-focused maps, which helped further refine the alternatives.

We also emphasized the importance of cross-cutting concerns that apply to every solution, and we discussed which concerns should be considered by solution architects. Remember that it might be too challenging to address all of the concerns on day one of your cloud journey. It is, therefore, interesting to think of different maturity levels, and how we would put them on a roadmap to manage our stakeholders' expectations.

Next, we highlighted the containerization components, with a focused map that depicted the container landscape of Azure. We also considered other dimensions, such as cost, complexity, and the level of residual operations of each option.

Lastly, we went through an exercise that consisted of building a solution architecture, with the help of the solution architecture map, when given a business scenario. We also walked you through a code sample that implemented this architecture.

In the next chapter, we will cover infrastructure-related aspects, such as monitoring, connectivity, and disaster recovery, which should help you complete the reference architecture we have built in this chapter. Later, we will provide more detailed coverage of AKS, one of the most popular services.

3

Infrastructure Design

In this chapter, we will focus on infrastructure architecture with Azure. Here, we will review the different concerns that every infrastructure engineer and architect has to deal with on a daily basis. More specifically, we will cover the following topics:

- The Azure infrastructure architecture map
- Zooming in on networking
- Zooming in on monitoring
- Zooming in on high availability and disaster recovery
- Zooming in on backup and restore
- Zooming in on HPC
- The AKS Architecture Map and a reference architecture for microservices

We will provide a 360° view of what it means to build infrastructure with Azure, including the most common practices and pitfalls. We will also see how challenging it is to have a consistent and coherent disaster recovery approach by walking you through a concrete real-world use case on a globally distributed API offering. Last but not least, we will dedicate a good part of this chapter to AKS, which has become a first-class citizen in many architectures, among which is the very popular microservices architecture. Reading this chapter will make you aware of some best practices and will help you gain insights into top-notch and trendy infrastructure topics in the Azure space.

Technical requirements

In this chapter, we will be using Microsoft Visio files. You will need Microsoft Visio to open the diagrams, although the corresponding PNGs are also provided.

The maps and diagrams used in this chapter are available at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/tree/master/Chapter03>.

The Azure infrastructure architecture map

The Azure infrastructure architecture map (as shown in *Figure 3.1*) is intended as your Azure infrastructure compass. It should help you to deal with the typical duties of an infrastructure architect, which we described in *Chapter 1, Getting Started as an Azure Architect*. Unlike the solution architecture map, which was more high-level, this map is a vertical exploration of infrastructure topics. It is by no means the holy grail, but it should help you to grasp the broad infrastructure landscape at a glance. Throughout this chapter, we will describe its various elements, and apply context using real-world implementations:

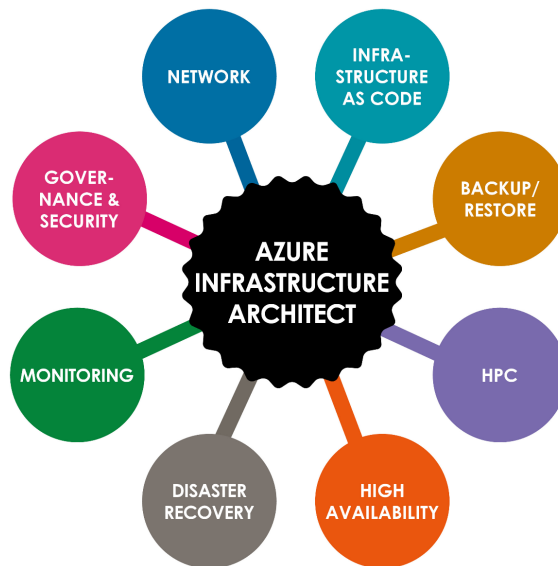


Figure 3.1 – The Azure infrastructure architecture map

Important note

To see the full Infrastructure Architecture Map (Figure 3.1), you can download the PDF file available at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/blob/master/Chapter03/maps/Azure%20Infrastructure%20Architect.pdf>.

The Azure Infrastructure Architect Map (Figure 3.1) has several top-level groups:

- **NETWORK:** This fundamental foundation is ubiquitous, and part of all the debates and trade-offs. Traditional IT is entirely based on the perimeter approach, which sometimes conflicts with the cloud's zero-trust approach. We will detail our options further in the *Zooming in on networking* section and will tackle the security-related bits in *Chapter 7, Security Architecture*.
- **MONITORING:** Once an application is built, we must run and monitor it.
- **GOVERNANCE/COMPLIANCE & SECURITY:** Governance and security are both key aspects of well-managed infrastructure. For the sake of brevity, we will explore this topic in *Chapter 7, Security Architecture*.
- **HIGH AVAILABILITY and DISASTER RECOVERY:** Every system/application might require some level of resilience. We have grouped these two top-level groups in the *Zooming in on high availability and disaster recovery* section.

- **BACKUP/RESTORE:** Azure has its own backup/restore mechanisms, which do not have much in common with traditional backup/restore mechanisms. We will explore this further in our *Zooming in on backup and restore* section.
- **HPC:** HPC stands for high-performance computing. Why not use the potentially limitless (except for our wallet) power of cloud infrastructure? We'll analyze the different HPC options in our *Zooming in on HPC* section.
- **INFRASTRUCTURE AS CODE (IaC):** IaC is such a large topic that we will explore it in *Chapter 4, Infrastructure Deployment*.

Now that we have highlighted the most important topics of our map, we will explore each of them, one after the other. Let's now focus on the networking aspects, to get acquainted with one of the most important infrastructure concerns.

Zooming in on networking

Networking is one of the essential foundations of any Azure landing zone. *Figure 3.2* shows the various connectivity options available in Azure:

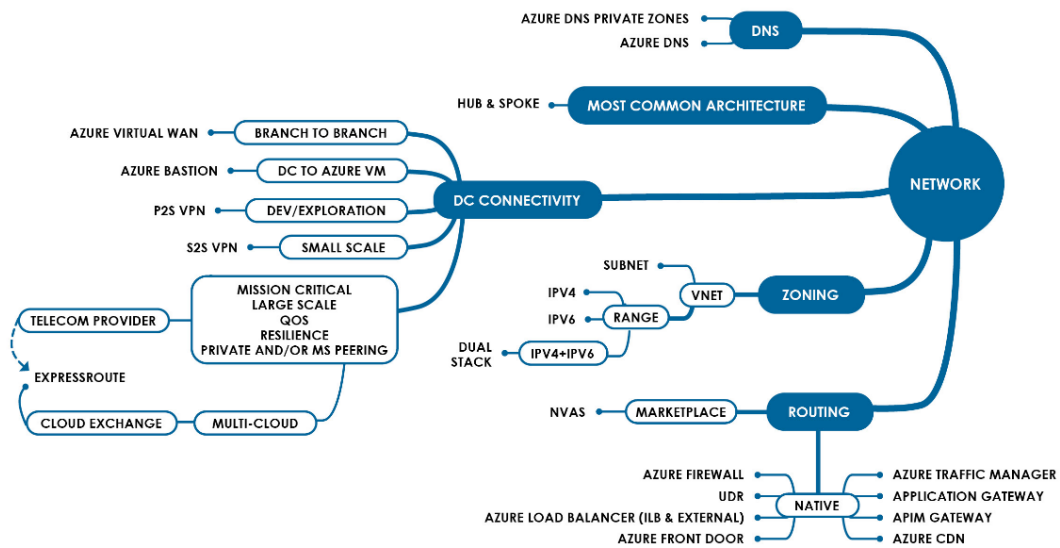


Figure 3.2 – Zooming in on networking

We introduced the landing zone concept in *Chapter 2, Solution Architecture*. We briefly explained that the purpose of a landing zone is to structure, govern, and rule the Azure platform for the assets that will be hosted on it. Controlling network flows is one of the key governance aspects. Controlling the network means mastering internal and external traffic, inbound and outbound, flow logs, and so on. This is a vast topic and an important challenge. Let's now dive deeper. The network section has five top-level groups:

- DNS
- MOST COMMON ARCHITECTURE
- DC CONNECTIVITY
- ZONING
- ROUTING

In the **DNS** section of *Figure 3.2*, we see two DNS services, which are public and private **DNS zones**:

- **Azure DNS zones** help you manage your public endpoints. In some situations, you might want to delegate the management of a single domain to an Azure public DNS zone. In doing so, you will create TXT, A, and CNAME records directly in Azure, instead of in your own DNS server. The benefit is, of course, to use a fully managed service.
- **Private DNS zones** help resolve privately addressable IP addresses. For example, when enabling **Azure Private Link** for a storage account, Azure makes use of private DNS zones to map a **record (of type A)** to the private IP of the storage account. They also create a CNAME record as an alias, between the public and the private endpoints. Private DNS zones are also used when setting up a private AKS cluster, which we will see in the *AKS Architecture Map and a reference architecture for microservices* section, as a way to map the API server name to its private VIP.

Note that **conditional forwarding** is currently not supported with private DNS zones, which leads to a situation where you need a self-hosted DNS server to route traffic between on-premises and Azure. This is usually achieved with the most common architecture, which we will describe in the next section.

The most common architecture

As we briefly described in *Chapter 1, Getting Started as an Azure Architect*, the **hub and spoke architecture**, presented in *Figure 3.3*, is the most frequent hybrid setup:

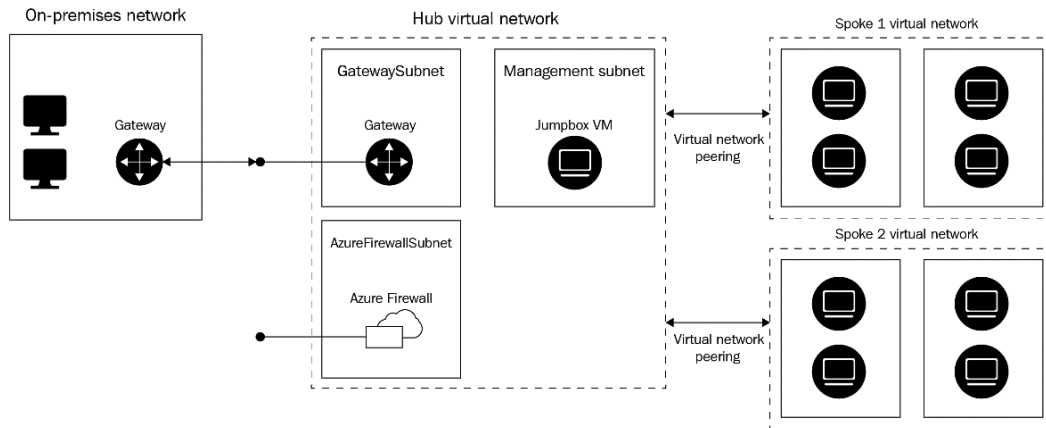


Figure 3.3 – Hub and spoke architecture

On the left-hand side of *Figure 3.3*, we have the on-premises network that is connected to the Azure Hub through either **ExpressRoute** or **VPN**. The Azure Hub is itself connected to the different spokes, which represent the assets deployed to Azure.

The **Azure Hub** is nothing but a regular **Azure Virtual Network (VNet)**. Its role is to route traffic between **spokes** and between the on-premises and cloud data centers. Each spoke, which is also a VNet, is peered to the Hub. VNet peering is non-transitive in Azure, meaning that if spoke 1 is connected to spoke 2 and 2 is connected to 3 (not present in the sample diagram), then 1 cannot talk directly to 3. Consequently, the hub and spoke architecture is used to simplify the structure and to have a central VNet that is connected to all the others.

Peering spokes directly is also possible, but it is usually not recommended because of the extra operational overhead. VNet peering is global, meaning that you can peer a VNet that sits in the West US region with another one sitting in the West Europe region, but beware that cross-regional peering incurs extra outbound traffic costs. Each spoke usually hosts the services of one single asset, while the shared services (DNS for instance, as highlighted in our previous section) are hosted in the hub.

Beware that it might take you several months to set up the hub and spoke architecture and to connect the Hub to the on-premises data center. This is often overlooked by customers, but this is by no way a trivial thing because it involves the network and security teams, which often have no or very little Azure expertise. Also make sure you consider all the services that cannot be created inside a VNet (Azure Functions on the consumption tier, serverless Azure Data Factory, Cognitive Services, and so on), which make them *de facto not fully* compatible with the Hub and Spoke architecture. This might be very confusing because most services can be private-link-enabled but this is not at all a full fit with hub and spoke. We will tackle, in detail, private connectivity and firewalls in *Chapter 7, Security Architecture*. The message we want to convey here is this: do not have a one-size-fits-all approach and think that you will have no more public endpoints or that you will control them all!

We will now look through our options to connect Azure to our data center.

Data center connectivity options

As shown in *Figure 3.2*, a **site-to-site (S2S)** VPN is mostly used in small-scale production workloads, meaning that the volume of traffic between on-premises and Azure remains low. With a VPN, you may have a ton of assets deployed in Azure with a low volume of interactions between both data centers. Volume is not the only thing to consider since the VPN option does not come with any SLA, in terms of resilience and latency. Azure supports many VPN protocols, such as **Secure Socket Tunneling (SSTP)**, OpenVPN, and IKEv2.

For mission-critical hybrid workloads, we recommend that you rely on **Azure ExpressRoute (ER)**. ER is fully supported by Microsoft, with guaranteed SLAs, such as an uptime of 99.95%. The bandwidth you have depends on your ER subscription, and it ranges from 50 Mbps to 100 Gbps for ExpressRoute Direct.

Connectivity between the on-premises data center and Azure is usually ensured by a telecom provider or a **cloud exchange broker (CEB)**, which consists of having a central connection point to a partner, such as Equinix. You would then get the traffic routed to the cloud provider, such as Azure, AWS, Salesforce, and so on. If you already have a connection to a CEB or a telecom provider, ER activation can be fast, or else it might take several months, depending on whether your data center is located near urban infrastructures or not. Besides an S2S VPN, you can also use a **point-to-site (P2S) VPN**, but this is mostly while starting your cloud journey (for example, as a temporary way to overcome the lack of a proper S2S or ExpressRoute setup).

Azure Bastion (AB) can also be used to connect on-premises machines to Azure-hosted virtual machines, without the need for public IP addresses for the VMs, nor a pre-existing S2S VPN or ExpressRoute. AB must be deployed its own dedicated subnet named *AzureBastionSubnet*. You can see it as a managed jump box, with a hardened public access point, from which you can connect to non-public VMs, using the Azure portal directly (a mere browser), instead of RDP/SSH. Traffic between the client and AB occurs over TLS, and the traffic is switched to RDP/SSH between AB and the target VM. Consequently, there is no need to expose ports 3389/22 publicly nor to open these ports in your on-premises firewalls. Without AB nor any other pre-existing private connectivity setup, your VMs must have a public IP to be accessible. If you do not want to pay for AB (or are still exploring Azure), you should at least enable **Security Center's JIT** (not to be confused with PIM's JIT) feature to limit the public exposure of your VMs. The purpose of JIT is to enable just-in-time incoming inbound traffic on demand and to block it after use.

Before effectively connecting data centers together, you need to think about your network zoning, and that is what we will discuss next.

Zoning

Network zoning is ensured by **VNets** and **subnets**. Each VNet is divided into different subnets that can communicate by default. When layering your VNets, make sure to calculate each subnet size carefully. When dealing with virtual machines only, it is rather easy to anticipate the number of IP addresses that will be required per subnet. This is not the same with Azure-native services, such as API Management, Application Gateway, AKS, and so on. You must also consider that Azure reserves five IP addresses for each subnet that is created.

You should always double-check Microsoft recommendations on the minimal required per-service subnet size, and you should consider scaling from the start. A subnet size cannot be altered, as long as some resources depend on it. So, you could quickly face issues in production, should your subnets not be large enough. Another thing to consider is that VNet address spaces should not overlap with each other. You may only neglect this aspect when working with isolated VNets that are not (and never will be) peered to other VNets, nor to the on-premises environment. Note that Azure VNets support both IPv4 and IPv6 address spaces, and they can even serve dual-stack systems. However, some services such as Azure Application Gateway still only work with IPv4. However, Azure services gradually increase their IPv6 compatibility, but keep their capabilities in mind when designing your networks. Once your VNet and subnet plumbing (address spaces, layering, and peering) are ready, you need to analyze your routing and firewalling options, and that is what comes next.

Routing and firewalling

The last top-level group of *Figure 3.2* is **routing and firewalling**. Azure makes use of **system routes**, which allow subnets to communicate, as well as peered VNets. Outbound traffic to the internet is also enabled by default, as well as publicly accessible workloads, such as virtual machines with a public IP. However, in a hub and spoke setup, subnets are associated with **user-defined routes (UDR)**, which override system routes, allowing them to ultimately forward the traffic to the Hub.

A common approach is to rely on a **network virtual appliance (NVA)** in the Hub, to rule the network traffic. Companies often purchase NVAs to keep using their existing software, such as Palo Alto, Check Point, and Citrix (to name a few). While this is understandable, none of these NVAs were initially designed for the cloud, and you must manage them entirely yourself.

An alternative option is to use **Azure Firewall**, a fully managed service that allows you to rule both inbound and outbound traffic. Feature-wise, Azure Firewall is still not at the same level as its marketplace competitors. One of the most in-demand features is an **Intrusion Detection System (IDS)/Intrusion Prevention System (IPS)**, which is still not supported at this time, although Azure Firewall can block some threats identified by **Microsoft Threat Intelligence**. If IDS/IPS is not a strong requirement (imposed by regulators), you should always favor Azure Firewall over NVAs, as it was conceived for cloud-native workloads, and Azure Firewall is highly available and scalable by design. Note that as of December 2020, there is a private preview going on for IDS/IPS with Azure Firewall, and we can therefore reasonably expect to have this feature generally available by 2021/early 2022.

Setting up the NVA, making sure it is highly available and that it is disaster recovery compliant, is rather hard and has a huge impact on costs. So, think twice before rushing to an NVA. Start from the actual requirements, not from the solution.

When dealing with web apps and APIs, you are also required to have a pure layer 7 **web application firewall (WAF)**. **Azure Application Gateway** is not only a reverse proxy, but it also can be enabled with out-of-the-box WAF policies to protect against the OWASP top 10. The policies can be tailor-made for your needs. Azure Application Gateway can be connected to both private and publicly accessible backends, as opposed to **Azure Front Door (AFD)**, which can currently only connect to public backends. Whenever possible, you should favor AFD, since it is a global service and has a **point of presence (POP)** everywhere in the world.

WAF policies can also be associated with AFD, and AFD encompasses static content caching, like a CDN appliance. One thing to note with AFD is that you should pay attention to its health probes. Since AFD is global, many of its POPs will run the health checks against your backends, which can result in between 200 and 1,200 calls per minute. Make sure that you configure the health probe correctly, so as to only return an empty response with a 200 result code and to avoid extra bandwidth costs. AFD has multiple routing methods: priority, latency, weighted, and session affinity, which enables many scenarios for globally distributed assets and disaster recovery mitigation measures.

Azure Traffic Manager (ATM) is also a global service, which is mostly used for geo-distributed workloads. Unlike AFD, ATM does not see the traffic going from the client to the backend, because it is based on DNS routing. For pure HTTP workloads, AFD should be preferred over ATM because it has some HTTP-friendly features, such as HTTP acceleration. AFD is also an all-in-one product, with reverse proxy and WAF capabilities.

Whenever you deal with API architectures, you should use an API management solution. **Azure API Management (APIM)** has recently joined the leader group of Gartner's Magic Quadrant for API management solutions. A typical topology for controlling traffic to APIs is to use a WAF in front of APIM and to use APIM's features to version and manage proxy APIs by targeting backend services. APIM can also be used for global deployments, and it has a built-in resilience mechanism to automatically fail over to available managed gateway units.

A common approach consists of using AFD in front of APIM's public VIP and using APIM policies to only accept the traffic coming from the AFD instance that has a unique ID. APIM allows you to deploy self-hosted gateways that are hosted as containers on-premises, in Azure or in another cloud. Both managed and self-hosted gateways are **policy enforcement points (PEPs)**, meaning that every request hitting the gateway must satisfy the policy requirements, in order to be forwarded to the actual backend service. A very common policy is the **throttling policy**, which is a very efficient way to prevent abuse and DOS/DDOS attacks. APIM's policy engine is very rich and can even contain C# expressions.

Controlling network traffic is very important; likewise, so is monitoring. So, let's now explore monitoring.

Zooming in on monitoring

Figure 3.4 is the same as the one we had in *Chapter 2, Solution Architecture*. In this section, we will explain a typical approach to **monitoring** Azure applications with native tools. The usage of Splunk, or any other third party, is beyond the scope of the book:

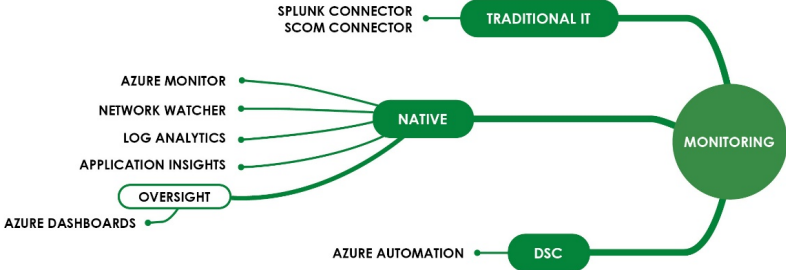


Figure 3.4 – Zooming in on monitoring

When an application is deployed to Azure, we must do the following:

- Monitor the application events. This can be achieved with **Application Insights**. Note that very recently, Microsoft launched **workspace-based Application Insights**, which in a nutshell couples Azure Application Insights and Log Analytics together.
- Monitor the Azure services, health. This can be achieved by redirecting diagnostic logs to **Log Analytics**.
- Define alerts on standard metrics or specific diagnostic log events.

Firstly, it is important to distinguish between logs and metrics. Log data can be used to perform root-cause analysis of a problem and/or to monitor the daily usage and to better understand the overall platform health. **Azure Monitor Metrics** provides a way to monitor specific service metrics and to define alerts when some thresholds are reached. Azure has the following different types of log data:

- **Activity logs:** They are common to every Azure service. They allow you to monitor the categories shown in the following figure:

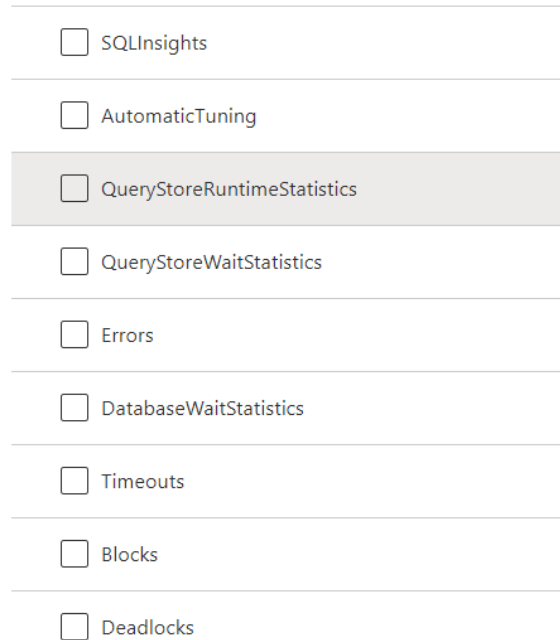
Category details

log
<input type="checkbox"/> Administrative
<input type="checkbox"/> Security
<input type="checkbox"/> ServiceHealth
<input type="checkbox"/> Alert
<input type="checkbox"/> Recommendation
<input type="checkbox"/> Policy
<input type="checkbox"/> Autoscale
<input type="checkbox"/> ResourceHealth

Figure 3.5 – Activity log categories

These logs are also interesting from a security perspective, as they keep track of who does what against the resource.

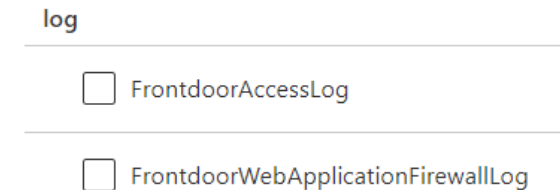
- **Diagnostic logs:** They can be service-specific or end up in the shared **AzureDiagnostics** log category. The following figure shows you an example of diagnostic logs for Azure SQL:



A screenshot of the Azure SQL diagnostic logs configuration interface. It shows a list of log categories, each with a checkbox. The categories are: SQLInsights, AutomaticTuning, QueryStoreRuntimeStatistics (highlighted), QueryStoreWaitStatistics, Errors, DatabaseWaitStatistics, Timeouts, Blocks, and Deadlocks. Each category is separated by a horizontal line.

Figure 3.6 – Azure SQL diagnostic logs

Meanwhile, AFD reports on the log categories in *Figure 3.7*:



A screenshot of the AFD log categories configuration interface. It shows a list of log categories, each with a checkbox. The categories are: FrontdoorAccessLog and FrontdoorWebApplicationFirewallLog. The word "log" is written above the list. Each category is separated by a horizontal line.

Figure 3.7 – AFD log categories

As you can see, they are very different from one service to another. You should always have a look at the specifics, in order to make sure you can control what you want to control. Most services offer the possibility to redirect logs to one or multiple targets, as illustrated in *Figure 3.8*:

Category details	Destination details
log	<input type="checkbox"/> Send to Log Analytics
<input type="checkbox"/> FrontdoorAccessLog	<input type="checkbox"/> Archive to a storage account
<input type="checkbox"/> FrontdoorWebApplicationFirewallLog	<input type="checkbox"/> Stream to an event hub
metric	
<input type="checkbox"/> AllMetrics	

Figure 3.8 – Sending diagnostic logs to a target repository

This makes it possible to centralize most service logs in Log Analytics for analysis and in Azure Storage for archiving. Event Hubs can be used to let third parties grab log data.

Once the data is in Log Analytics, it is possible to perform advanced queries and even to define alerts on query results. *Figure 3.9* shows an example that uses **Kusto query language (KQL)** to detect AFD's firewall security events:

The screenshot shows a Kusto query editor with the following query:

```
1 AzureDiagnostics
2 | where Category == "FrontdoorWebApplicationFirewallLog"
3 | project details_msg_s
```

Below the query, the results are displayed in a table. The table has a column named 'details_msg_s'. The results show four entries, all of which are 'SQL Injection Attack Detected via libinjection'. The interface includes a 'Run' button, a 'Time range' dropdown set to 'Last 30 minutes', and options to 'Save', 'Copy link', and 'New alert rule'. The results section shows 'Completed. Showing results from the last 30 minutes.' with a duration of '00:00:00.383' and '106 rec'.

Figure 3.9 – AFD firewall logs

Figure 3.9 shows that some SQL injection attacks were detected by AFD. KQL can also be used to render charts. The next figure shows you how to render charts:

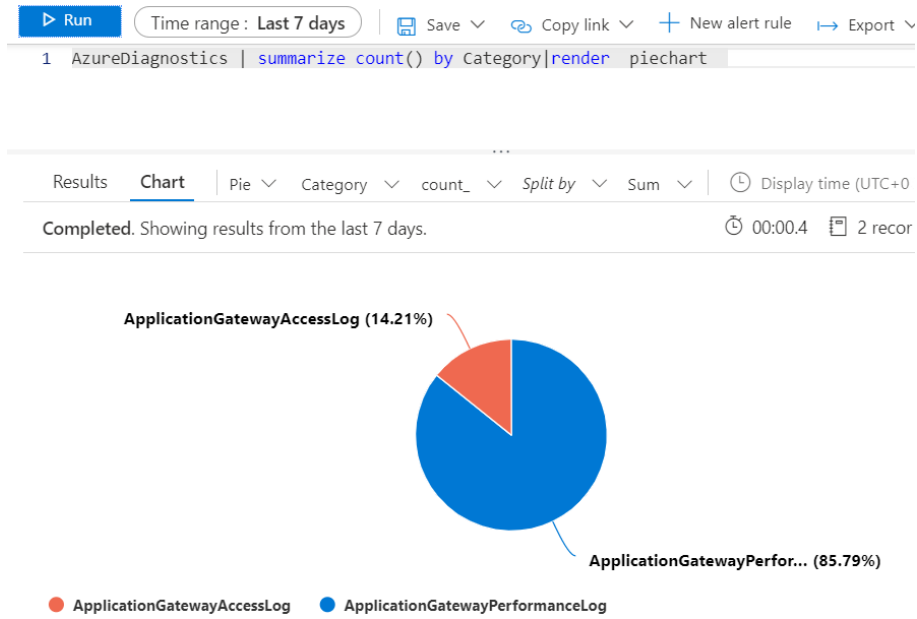


Figure 3.10 – Rendering charts with KQL

Such charts can easily be pinned to **Azure dashboards**, and new alert rules can be defined against KQL queries. When it comes to pure metrics, Azure Monitor Metrics is the easiest way to keep track and to define alerts when a given threshold is reached by a service. For example, Figure 3.11 shows you how to define an alert on AFD's backend latency:

Alert logic Monitoring 1 time series (\$0.10/time series)

Threshold ⓘ

Static Dynamic

Operator ⓘ Aggregation type * ⓘ Threshold value * ⓘ

Greater than Average 2000

milliseconds

Condition preview

Whenever the average backend request latency is greater than 2000 milliseconds

Evaluated based on

Aggregation granularity (Period) * ⓘ Frequency of evaluation ⓘ

15 minutes Every 5 Minutes

Figure 3.11 – AFD backend latency

This figure shows the definition of an alert that should fire whenever the average backend latency is over 2,000 milliseconds over an aggregated period of 15 minutes. This evaluation happens every 5 minutes. Such alerts are bound to one or more **action groups**, which are entities that define who to notify and how to handle the event. Notifications range from mere emails to SMS/voice messages. The associated actions can be hooked to many different services and systems, as illustrated in the following figure:

Create action group

Basics Notifications **Actions** Tags Review + create

Actions

Configure the method in which actions are performed when the action group triggers associated details, and add a unique description. This step is optional.

Action type ⓘ	Name ⓘ
ITSM ^	<input type="text"/>
Automation Runbook	<input type="text"/>
Azure Function	
ITSM	
Logic App	
Secure Webhook	
Webhook	

Figure 3.12 – Action group action types

You can automate the alert handling, using any of the Azure services present in *Figure 3.12*. Webhooks are a way to reach out to any system, such as, for instance, Dynatrace, which has an Azure integration module. ITSM is certainly a very interesting option, as it allows you to create a ticket in your preferred ITSM tool, such as ServiceNow. If you combine KQL with the pre-defined metrics and alert mechanisms of Azure, you have a very powerful monitoring system. On top of that, you can always make use of Event Hubs to let third-party tools gather all the log data. Monitoring Azure components using native services is rather trivial, unlike our next topic: high availability and disaster recovery.

Zooming in on high availability and disaster recovery

First of all, let's review the difference between **high availability** and **disaster recovery** and put that in the Azure context. A **high availability (HA)** solution is continuously available for a desired amount of time. In Azure, most HA solutions are scoped to a single geographical region.

Disaster recovery (DR) aims to recover from a severe incident, such as a fire (or flooding) in the data center, an earthquake, or any other type of heavy damage. In Azure, an example of a severe outage is the complete unavailability of an entire region, or of a service within a region. DR-compliant systems often rely on multiple regions, which incurs extra costs. Usually, a design that is DR-compliant is also HA.

Whether you design a solution for HA or DR depends on the expected **recovery time objective (RTO)** and **recovery point objective (RPO)** defined by the business or expected by your customers (if you provide the service). *Figure 3.13* is a zoom-in on high availability in Azure:

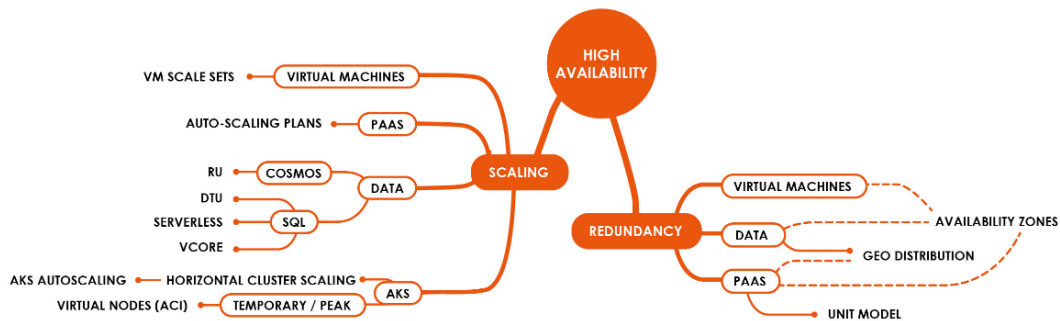


Figure 3.13 – Zoom-in on high availability

Figure 3.13 has two top-level groups:

- **REDUNDANCY:** Uses Availability Zones, geo-distribution, and/or a unit model
- **SCALING:** Uses VM scale sets, autoscaling plans, serverless, horizontal cluster scaling, AKS autoscaling, and/or virtual nodes (ACI).

Scaling out (horizontal scaling) is a way to increase HA. Indeed, having multiple instances of a service or virtual machine will ensure greater availability should one instance become unhealthy. **Azure virtual machine scale sets** let you define a group of virtual machines that can scale out or down, according to the demand. They are also used by AKS as a way to scale the K8s cluster. Special care has to be taken with the services that run on the virtual machines for which you are entirely responsible.

Most PaaS services rely on **autoscaling plans**. In a nutshell, these plans let you define some thresholds, typically at the memory and CPU level, to scale out or down the underlying service. Regarding data stores, such as Azure SQL and Cosmos DB, you can rely on autoscaling DTU/vCore or RU within defined boundaries, in order to prevent a cost explosion.

In a non-production environment, you should use fixed DTU/vCore/RU to identify poor query patterns, as early as possible, and to prevent bad (cost) surprises in production. **AKS** relies on horizontal cluster scaling, based on virtual machine scale sets. **Virtual nodes** can also come to the rescue to add extra power, when needed, and to respond faster to a fast-growing demand.

REDUNDANCY is another way to increase HA. Azure Availability Zones are different physical data center locations within the same Azure region. With Availability Zones, you do not merely increase the number of instances of a service or machine, but they also span multiple physical infrastructures, while still remaining within a single region.

In the past, Availability Zones were mostly used in IaaS with virtual machines, but they have become more and more available for PaaS services as well. An example of this is the recent general availability (06/2020) of Azure Storage **GZRS (Geo-zone-redundant storage)**. **Geo distribution**, or replication, is a way to achieve both HA and DR at the same time. Most data services have geo-distribution capabilities, with multiple read or read/write regions. Finally, there is what we call the **unit model**, which is a way to increase the number of units a given PaaS or FaaS service has to run. For example, Azure API Management gateways come with the unit concept, where you pay per unit. The same thing applies to Stream Analytics jobs and other services.

Next is the disaster recovery piece, as illustrated in *Figure 3.14*:



Figure 3.14 – Zoom-in on disaster recovery

Admittedly, *Figure 3.14* is far from being comprehensive. There are so many Azure services that an entire map on DR would probably not suffice. We simply wanted to highlight some usual suspects, such as virtual machines, databases, HTTP-based workloads, and messaging workloads. **Azure Site Recovery** is the most important tool to ensure DR for virtual machines, between on-premises and the cloud and one or more Azure data centers. Database engines rely on the geo-distribution that we have just explained. **Azure Service Bus**, **Event Hubs**, and **Event Grid** only replicate metadata (queues, topics, and subscriptions), not the message themselves. That is why a full DR-compliant solution requires some client logic as well. The latter example illustrates the complexity of designing a coherent end-to-end DR-compliant solution, because not all services have equivalent DR features, while most solutions rely on a mix of services. HTTP-based workloads are often deployed to multiple regions and appliances, such as **Front Door** or **Traffic Manager**, which handle the routing and DR logic.

To go a bit further with DR, let's go through a real-world scenario and craft a possible architecture:

Contoso wants to propose globally distributed APIs to their customers. They span three continents: North America, Europe, and Asia. The SLA of their customers is very demanding, as they want to be up and running 99.99% of the time. They want to optimize response times and have multiple POPs so that those POPs that are the closest to customers handle the requests. Given the SLA, the solution should survive a regional outage.

The next diagram shows a possible architecture to respond to these requirements:

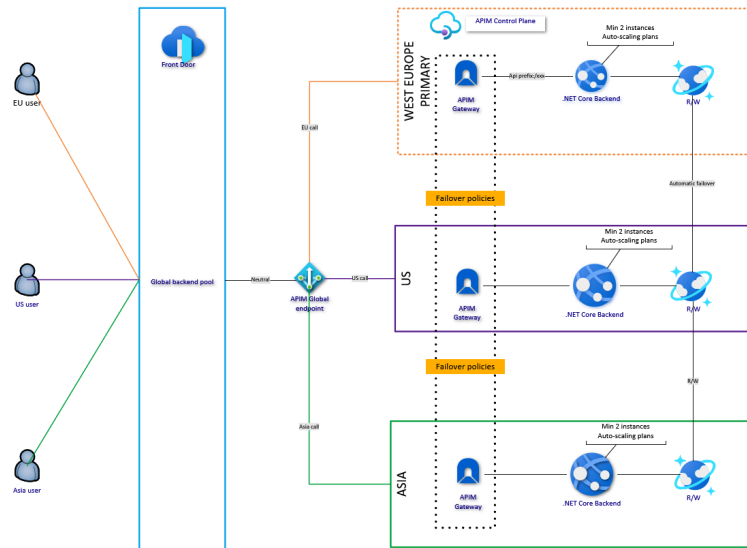


Figure 3.15 – An active-active DR-compliant global API offering (diagram available on GitHub)

On the left-hand side, there are end users or end user applications that call our APIs. AFD is a global autoscaling service that has many POPs worldwide. Whenever possible, try to use global services, as they are DR-compliant out of the box. In our design, Front Door ensures the POP, as well as the layer 7 WAF, before customer traffic hits the APIs. Front Door's backend pool is the global Azure **API Management (APIM)** gateway endpoint.

Only the APIM premium tier is multi-region-aware. Our design has one gateway unit in Europe, one in the US, and one in Asia. APIM's global load balancer routes calls to its closest gateway unit by default, so this is in line with our scenario. APIM is also able to detect whether a gateway is healthy or not. In case a gateway (say Europe, for example) is not healthy, then it will route the traffic to the other remaining gateways, hence the reason why we take a single unit per region. Having multiple gateway units per region is also possible, but it will drastically increase your costs.

We must write APIM failover policies to route the traffic to our backend services because the global APIM load balancer is unaware of the actual backend health. These backends are hosted on Azure App Service and run multiple instances (a minimum of 2). They rely on autoscaling plans to handle peaks. Lastly, we use Cosmos DB with multiple write regions, and we have activated Cosmos's automatic failover. Having multiple write regions allows for zero downtime should a regional outage occur, since the system is already in active-active mode. An alternative to APIM policies is to monitor the actual backend health via AFD directly and let AFD handle failover concerns. While it is possible, it may not be easier.

The top-right dotted rectangle (*Figure 3.15*) is our primary region, because only a single premium instance of APIM is deployed to **West Europe (WE)**. This means that if WE goes down, everything will still be up and running, except APIM's control plane, which will prevent any modification of the APIM instance, as long as the primary region is not back up and running. This has no impact on the customer experience. Even if only one Cosmos DB goes down, our .NET Core backend is smart enough to connect to the remaining databases, thanks to the built-in features of the Cosmos DB SDK.

However, our architecture has some potential drawbacks and might not be suitable in every situation:

- Its active-active geo-distributed database relies on **eventual consistency**. This means that data retrieved by customers might not be the most recent. Indeed, by default, it is not possible to enable strong consistency for regions that are separated by more than 5,000 miles. This can be requested from Microsoft, but the impact on write latency is huge.

- We have 3 gateways, meaning ~7500 euros/month, just for APIM. We could have self-hosted the gateways (~3 times cheaper) but then we would lose APIM's SLA for the data plane, and we still need to foresee costs and HA to host the gateways.

Should you add other services to the mix (such as Service Bus, Event Grid, and Redis Cache, to name a few), you will have to make sure to keep a consistent HA-DR story. In complex architectures, you might have to plan for a degraded mode to keep costs and complexity low. As a rule of thumb, a DR-compliant architecture costs about 2.5 times more than a single region-scoped architecture. Why 2.5 and not 2? Simply because only the premium tier of most PaaS services ships with DR features, while you can rely on standard tiers (which are cheaper) when working with a single region.

In this section, we mostly addressed the RTO objective, not the RPO one, which can partially be addressed by a proper backup and restore strategy, our next topic.

Zooming in on backup and restore

Backup and restore processes are also part of the broader disaster recovery picture. However, you might end up with a corrupted database or accidental data deletion, even in a non-disaster situation:

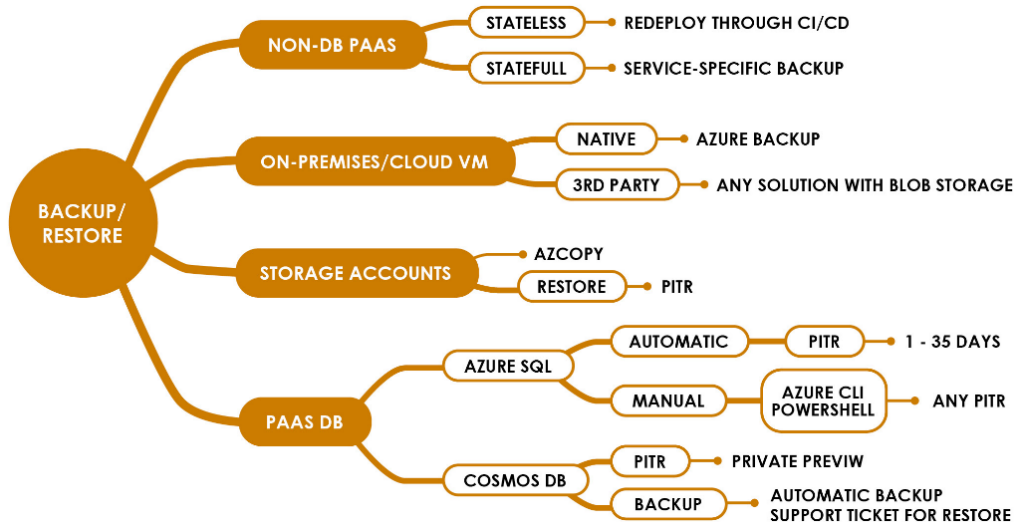


Figure 3.16 – Zoom-in on backup and restore

Figure 3.16 is far from being comprehensive, but it should give you the key aspects to consider for a good backup and restore strategy. *Figure 3.16* includes four top-level groups:

- **NON-DB PAAS:** This top-level group refers to managed services that are not related to database engines.
- **ON-PREMISES/CLOUD VM:** This is applicable to both on-premises and cloud-hosted virtual machines.
- **STORAGE ACCOUNTS:** AzCopy is usually used to push and pull data to/from storage accounts.
- **PAAS DB:** This top-level group relates to database-specific managed services.

We first distinguish database services from other PaaS services, because the way to back up and restore them is totally different. In the **NON-DB PAAS** group, we have **stateless** and **stateful** services. An example of a stateless service is Azure Functions. With such services, a modern way to restore them, if an issue occurs, is to redeploy them through CI/CD, thanks to the IaC pipelines.

With stateful services (such as Azure API Management, where API consumers might have been onboarded through the development portal), we also have to consider the inner state of the service, which in this case are the subscribers. Redeploying through CI/CD would not be okay, as we would lose the subscribers. With such services, you can rely on the service-specific backup/restore capabilities. Since there are too many different ways of doing this, we will not dive into it in this book – just keep in mind that service-specific backup/restore strategies exist, and you need to look at them individually.

For **virtual machines**, you can virtually use any backup solution. Today's vendors have connectors to Azure Blob Storage to store both on-premises and cloud virtual machine disks. The native offering is **Azure Backup**, which can also be used for both on-premises and cloud-based machines. Regarding **storage accounts**, you can explicitly take backups with **AzCopy**, a command-line tool that lets you back up both blob and table storage. It does not offer a true restore feature, because the way to restore blobs is simply to copy the backed up ones to another storage account or the same storage account, by overwriting them. It's interesting to note that storage accounts have a **point-in-time restore (PITR)** feature that allows you to recover from data corruption or an accidental deletion, although the latter can be mitigated by simply enabling the **soft delete** feature. PITR is not enabled by default, so pay attention to enable it should your storage block blobs be of high importance.

The **PAAS DB** group shows the possibilities for **Azure SQL** and **Cosmos DB**. Azure SQL takes automatic backups for 7 days by default. The **PITR (point in time recovery)** can be between 1 and 35 days. It is also possible to take backups manually, using command-line tools, and to restore them at any time. Cosmos DB is different, because the only way to restore a database or container is to reach out to Microsoft support. There is, at the time of writing, a private preview program to let you test PITR with Cosmos.

When designing a solution, make sure to verify that the expected RPO is in line with Azure's capabilities.

Sometimes, some workloads go beyond typical resource needs, and require much more power. HPC, our next topic, is a possible answer to this.

Zooming in on HPC

High-performance computing (HPC) is a pure infrastructure topic, because it boils down to bringing an unusual amount of compute and memory to a given workload. In general, HPC jobs are handled by dozens, hundreds, or even thousands of machines in parallel. *Figure 3.17* shows most of the current Azure HPC landscape:

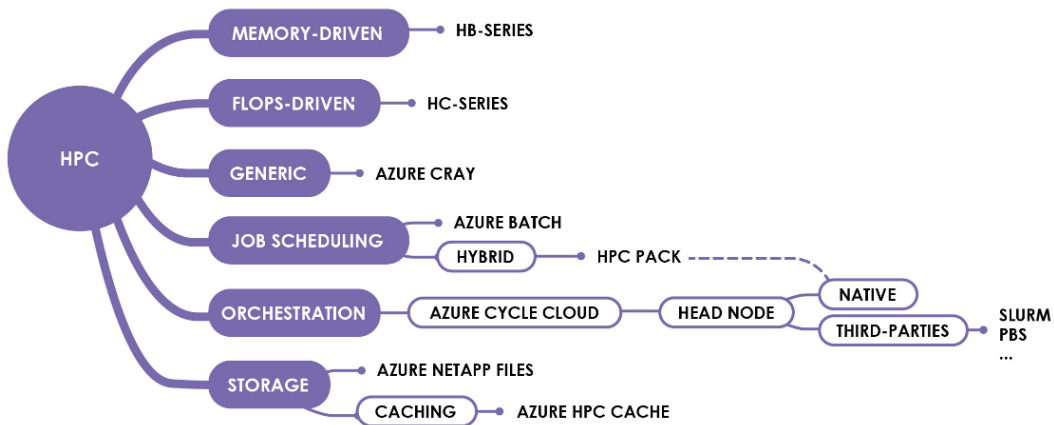


Figure 3.17 – Zoom-in on HPC

For memory-driven workloads, such as **computational fluid dynamics (CFD)**, you may rely on **HB-series** virtual machines, which are bandwidth-optimized. For **FLOPS**-driven (short for **floating-point operations per second**) workloads, which require a fast and optimized CPU, you can rely on the **HC series**. If you are unsure of whether your workload is memory- or flops-driven, you might rely on **Azure Cray**, a supercomputer delivered as a managed service. When it comes to job scheduling and underlying infrastructure management, you can count on **Azure Batch** and **Azure Cycle Cloud**. Azure Batch is a fully managed service that abstracts away the underlying infrastructure. Azure Cycle Cloud is more interesting if you have to integrate with third-party HPC solutions. In terms of storage, which plays a crucial role in HPC solutions, you can work with Azure NetApp Files or Azure HPC Cache, or both at the same time. Azure NetApp Files brings massive capabilities, in terms of I/O, while Azure HPC Cache is suitable in read-heavy scenarios.

We have now covered most of the Azure Infrastructure Map. The time has come to explore a very in-demand service in more depth: AKS. It comes with its own practices and infrastructure specificities.

AKS infrastructure

AKS is an entire world within the Azure universe. This is by no means a service like the others. It is a partly managed service, as shown in *Figure 3.18*.

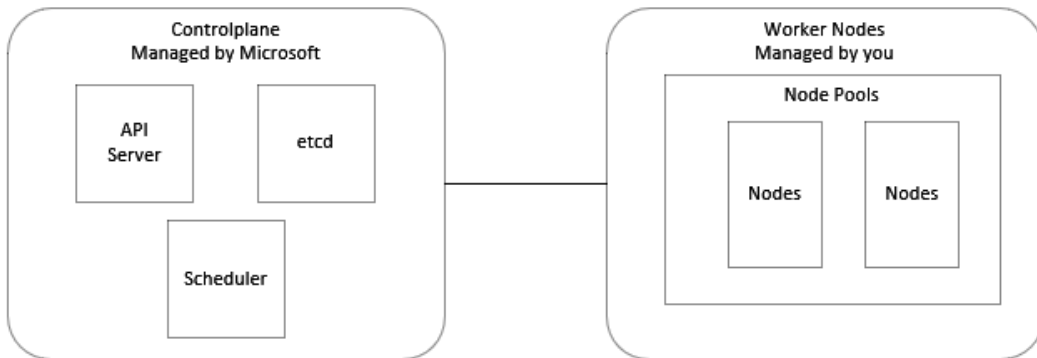


Figure 3.18 – AKS, a semi-managed service

The control plane is the brain of AKS, and it is fully managed by Microsoft for free. Your duty as an Azure infrastructure architect is to take care of the worker nodes, which are plain virtual machines, connected to the brain via **kubelet**, the **Kubernetes (K8s)** primary node agent. It runs on each node, and the agent registers the node with the API server automatically. Rest assured, Azure comes with pre-defined node images, and you do not have to build the worker nodes yourself, just manage them. Although self-hosting a **K8s** cluster is even more demanding, you should not neglect the number of operations left to the cloud consumer when working with AKS. Unlike a fully managed PaaS or FaaS service, AKS requires special care and upfront analysis before usage.

AKS surely deserves an entire book for itself, so we will not be able to cover all the bits and bytes, but we will try to highlight the most important aspects to consider when starting an AKS journey in the following sections:

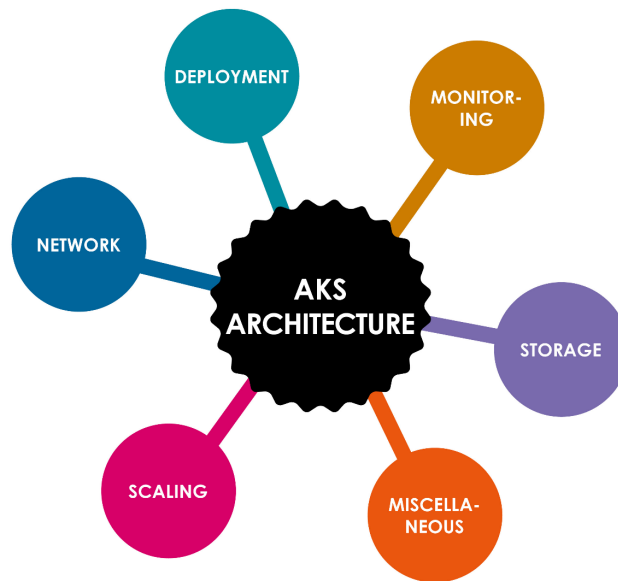


Figure 3.19 – The AKS Architecture Map

Important note

To see the AKS Architecture Map (Figure 3.19) in full size, you can download the PDF here: <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/blob/master/Chapter03/maps/AKS%20Architecture.pdf>.

The AKS Architecture Map has six top-level groups:

- **DEPLOYMENT:** AKS supports all the possible deployment models to ensure zero downtime and continuous delivery. Beyond deploying applications themselves, we will explore segregation between assets in our *Zooming in on deployment options with AKS* section.
- **NETWORK:** As for regular Azure infrastructure, networking is a very important pillar. AKS has some specific characteristics that we will highlight in our *Zooming in on networking options with AKS* section.
- **MONITORING, STORAGE, SCALING, and MISCELLANEOUS:** For the sake of brevity, we will not detail each and every top-level group. We will simply leave the *zooming in* maps for your reference, with a high-level description, and rather go through some concrete AKS use cases, mostly around microservices architectures, which is one of the top reasons to use AKS.

Let's start with AKS networking, probably the most challenging infrastructure topic.

Exploring networking options with AKS

As always, **networking** is the fundamental necessary evil for getting anything to work! Let's explore this map further here:

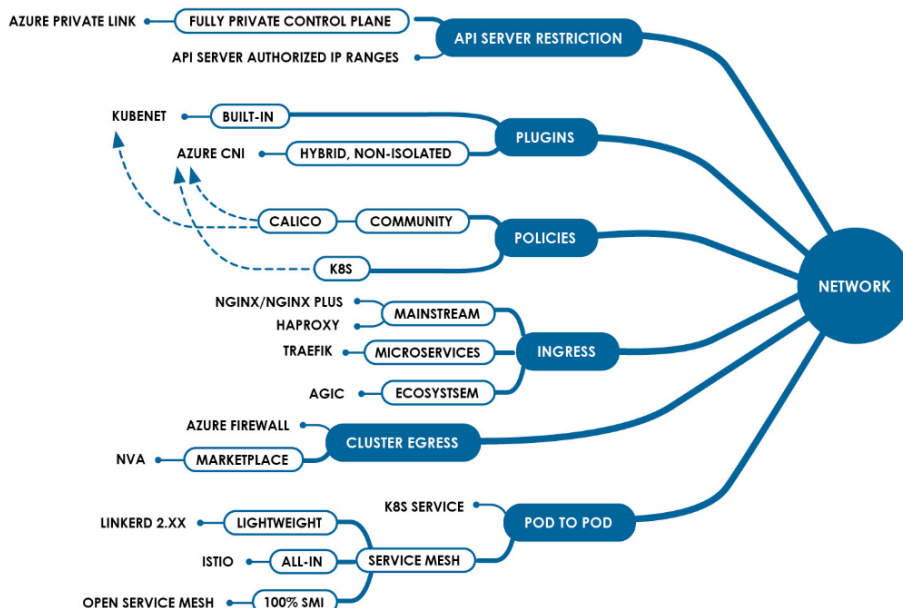


Figure 3.20 – Zoom-in on AKS network options

Figure 3.20 has six top-level groups:

- **POD TO POD:** Applicable to inter-pod network control.
- **INGRESS:** Applicable to cluster ingress controllers of AKS.
- **EGRESS:** Applicable to cluster egress. Pod egress is tackled in the **POD TO POD** section.
- **POLICIES:** A way to enforce network policies within AKS.
- **PLUGINS:** Network plugins work very differently in AKS. They have a dramatic impact on available features as well as on the method for allocating IP address spaces.
- **API SERVER RESTRICTION:** This section is about controlling who can talk to the API server.

POD TO POD shows that the most important options that communicate between each other are the **K8s Service** and a **service mesh**. A K8s Service is an out-of-the-box layer 4 component that supports TCP, UDP, and SCTP. This component behaves as an abstraction layer between the different pods, because pod IP addresses are constantly subject to change. Service meshes, which we will describe further later, are layer-7-aware and allow for smart load balancing (based on actual backend latency), mTLS, and for understanding typical layer 7 protocols, such as HTTP and gRPC. There are currently two market leaders: **Istio** (<https://istio.io/>) and **Linkerd** (<https://linkerd.io/>).

Istio is an all-in-one product that allows you to deal with network policies, resilience testing through fault injection, and advanced deployment techniques. It also offers greater observability over the activity of the pods deployed within a cluster. Linkerd is a simpler, lightweight service mesh that also offers mTLS, basic traffic split, and greater observability. Compared to Istio, Linkerd has a smooth learning curve and is very fast.

Istio has a visible impact on performance, and it comes with a steep learning curve. Open Service Mesh is still in its early days, but it aims at satisfying 100% of the **Service Mesh Interface (SMI)**, which makes it very open. This is clearly a product to keep an eye on in the coming months. Service meshes still rely on K8s services, not to route the traffic through the service, but rather to gather the pod IP addresses.

The next group from *Figure 3.20* is **EGRESS**. Egress is opened by default, meaning that AKS lets pods talk to the internet. There are a few ways to control this. First, you can add **Azure Firewall** to the mix, in order to control cluster-level egress, which is called egress lockdown. Second, you can use Istio's egress gateways to get pod-level egress control. In the case of a global hub and spoke setup, you can have **user-defined routes (UDRs)** to route the traffic to the Hub and to filter what is allowed and what's not from the Hub's NVA (when using an NVA). Note that these techniques are not mutually exclusive.

Let's now explore the **INGRESS** group. By default, services deployed into the cluster, and that have a ClusterIP, are not accessible from outside the cluster. To expose them to the external world, you need an ingress controller. Azure Application Gateway's AGIC component can be used to do the job. AGIC became generally available by the fall of 2019. It is not yet as feature-rich as other players, such as Nginx and Traefik, but the value proposal is to delegate the most operations to Microsoft's services. Nginx is globally used and is known for its good performance. Traefik has been designed with the microservices architecture in mind, which has a lot of moving parts and deployments. Traefik's configuration is very dynamic and does not require downtime or refactoring.

Our next group is **POLICIES**. Network policies in K8s are layer 4 policies, which can be seconded by service mesh layer 7 policies. K8s policies are not available when you're using K8's default network plugin, called **Kubenet**. In such a situation, you can rely on **Calico** to do the job. Project Calico is open source. Another option is to use Azure's default CNI plugin with the default K8s policies, or with Calico, since it is also supported. Opting for Calico also depends on other factors, such as also using Istio (for which Calico has built-in support). Whatever option you choose, the purpose of network policies is to control internal cluster traffic.

Let's now look at the **API SERVER RESTRICTION** group. As highlighted earlier, AKS includes a control plane that ships with an API server. The API server handles all the configuration options of a cluster. For connectivity, the API server is exposed to the internet, by default. There are two options to mitigate this: using network ACLs through IP restrictions, or using a fully private API server through the use of **Azure Private Link**. This causes a bit of an operational overhead, which we'll describe later in this chapter.

The time has come to explore one of the most important groups: **PLUGINS**. The network plugin you choose has a dramatic impact on capacity and scalability. When starting your AKS journey, you might want to make your life easier and use **Kubenet**, hence the reason why it is under the *dev/lab/dedicated/test/standalone* branch. It does not mean that Kubenet cannot be used in production, but in that case, it should be combined with Calico, to compensate for the lack of native network policy support. Upon creating your cluster, you have to choose between Kubenet or advanced networking, which is also known as **Azure CNI**. There are some significant differences between them, but the main difference is how IP addresses are allocated to pods.

When using Kubenet, IP addresses are only allocated to worker nodes, while pod IPs are NATed. So, each virtual machine that is part of the cluster gets an IP address, and each pod is proxied by the node's primary IP address. When working with Azure CNI, every worker node and every pod gets a dedicated IP. Therefore, Azure CNI requires many more IPs and a bigger subnet size. It is crucial to take this into account in your network design.

Figure 3.21 is a comparison table between Kubenet and Azure CNI on different aspects:

	Azure CNI	Kubenet
Subnet size	Considers both worker nodes and pods	Considers worker nodes only, meaning that a small subnet range might be enough
Autoscaling	Takes a margin when sizing the subnet, to allow both cluster growth and pod count growth	Considers the worker nodes growth only
Cluster upgrade (details follow)	Depends on the upgrade strategy, but in general, it requires more IPs	Depends on the upgrade strategy
Network policies (details follow)	K8s policies supported out of the box	Requires a plugin, such as Calico
Performance	Faster by default because no NAT (Network Address Translation) takes place	Slower by default because of NAT, but it can be mitigated with Calico
Virtual Kubelets (ACI in Azure)	Supported	Not supported (at the time of writing)

Figure 3.21 – Kubenet versus CNI

As you should have understood by now, if you opt for Azure CNI, you should carefully consider both cluster scaling and upgrading, to make sure you do not get blocked over time in production by a lack of available IP addresses in the AKS subnet. Microsoft provides some guidance on that topic at the following URL: <https://docs.microsoft.com/azure/aks/configure-azure-cni>.

At the time of writing, virtual Kubelets, which are called **Azure Container Instances (ACI)** in Azure, are only possible with Azure **Container Networking Interface (CNI)**. You should always enable them whenever possible, because they are free of charge and do not cause any more complexity. They are the serverless part of AKS, bringing you a serious amount of extra power, as shown by Figure 3.22:

```
Capacity:
cpu:      10k
memory:   4Ti
nvidia.com/gpu: 100
pods:     5k
```

Figure 3.22 – Output of the `kubectl describe node virtual-node-aci-linux` command

As you can see, virtual nodes represent solid extra capacity: 4 TB of memory, 10,000 CPUs, and up to 5,000 pods. So, even if you have not identified any use case for them yet, make sure to enable them because they come at no extra cost and cannot be added post-cluster creation. They should be useful sooner or later. Some typical use cases include (but are not limited to) the following:

- Resource-intensive jobs.
- Message or event handlers, since they are often ephemeral, which goes well with ACIs.
- CI/CD self-hosted ephemeral agents. These are particularly useful when running integration tests.

Cloud-native platforms are all based on a scale-out story, meaning that you're multiplying the number of instances. That usually works well, but sometimes you really need to allocate more power to a single pod, and that is where virtual kubelets come in handy. Instead of spinning up an entire worker node to scale out your pods, you may rely on serverless ACIs, where you only pay for the time they run. Do you remember our solution architecture use case, where we used ACIs instead of plain virtual machines (for cost-saving reasons)? We are in the same situation here.

As with regular ACIs used outside K8s, there is a delay before an ACI gets started, so your use case must tolerate it.

Note that the network plugin cannot be changed once the cluster has been provisioned, so be sure to make the right choice from the start. Once the cluster network is under control, we can start deploying assets to it. This is our next topic.

Exploring deployment options with AKS

Deployment has become a very central topic recently. Today's business lines expect frequent releases and no downtime. They also expect feature-level deployment and advanced testing mechanisms, such as A/B testing. *Figure 3.23* depicts the deployment options in AKS:

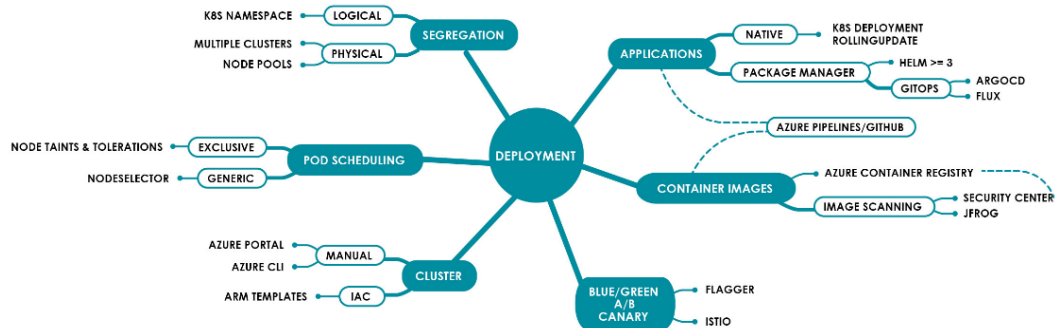


Figure 3.23 – Zoom-in on deployment

Figure 3.23 has six top-level groups:

- **CLUSTER:** Applies to everything that helps deploy clusters.
- **POD SCHEDULING:** Applies to everything that impacts how pods are scheduled.
- **SEGREGATION:** Applies to everything that helps segregate assets.
- **APPLICATIONS:** Tools and techniques that help deploy applications.
- **CONTAINER IMAGES:** Every container relies on an image that needs to be stored in a registry, with possible security scanning steps.
- **BLUE/GREEN – A/B – CANARY:** Various deployment models supported by AKS.

Before deploying any application, we need to deploy the cluster. The **CLUSTER** group depicts different options. In terms of IaC, ARM templates can be used to deploy the cluster. It is also possible to leverage the Azure CLI or the Azure portal to deploy the cluster. Typically, your CI/CD pipelines would use ARM templates, but in an exploration phase, you might simply use the portal.

A fundamental question is *how do we segregate assets?* That is where our **SEGREGATION** group comes in handy. If the asset is big enough, a dedicated cluster might be the best option. Note that a minimal topology for production consists of having three worker nodes.

If you share a cluster for multiple assets, you should use the K8s **namespace** resource type to segregate them. The namespace is a logical boundary that is heavily used in K8s. Note that even with a single asset, you might want to use the namespace to split the different layers (frontend, backend, and data) if you work with a layered architecture. Whether you use one or more clusters, you also have to think about your **node pools**. Each node pool is a series of worker nodes that are based on a virtual machine size. For instance, you might want to have a specialized node pool for databases, should you host them within the cluster, and another one for services.

Node pools have a direct relation with the **POD SCHEDULING** group, since deployments may influence where a pod should be scheduled by the K8s scheduler, by using **taints** and **tolerations**. If you deploy a database, you might first taint your database nodes with the taint `database=true` and then add tolerations to your database pods, to make sure your pods are scheduled onto the database-specific node pool. The default behavior of the K8s scheduler is to deploy the pod to the first available untainted node.

In AKS, every application requires at least a container image. That is what is covered by our **CONTAINER IMAGES** group. AKS can work with any container registry but **Azure Container Registry (ACR)** is probably the best default choice. When it comes to scanning (for security) the container images, ACR relies on Security Center's Qualys integration. An alternative is to use JFrog's container registry, which also includes image scanning.

Last but not least, we have the **BLUE/GREEN – A/B – CANARY** group. As explained earlier, these deployment techniques ensure frequent deployments and low to zero downtime. To leverage any of them, you can rely on **Flagger** and **Istio**. Linkerd also has a traffic split feature, but it's not as rich as Flagger and Istio's versions.

As with any other systems, deployed assets, as well as AKS itself, must be monitored to ensure a smooth user journey. Let's explore some of the AKS specifics when it comes to monitoring.

Monitoring AKS

As discussed earlier, for regular Azure services, **monitoring** is as important as developing an application. AKS has some specific tools and services to ensure proper monitoring.

Figure 3.24 shows the most important and frequently used monitoring and health check services for AKS:

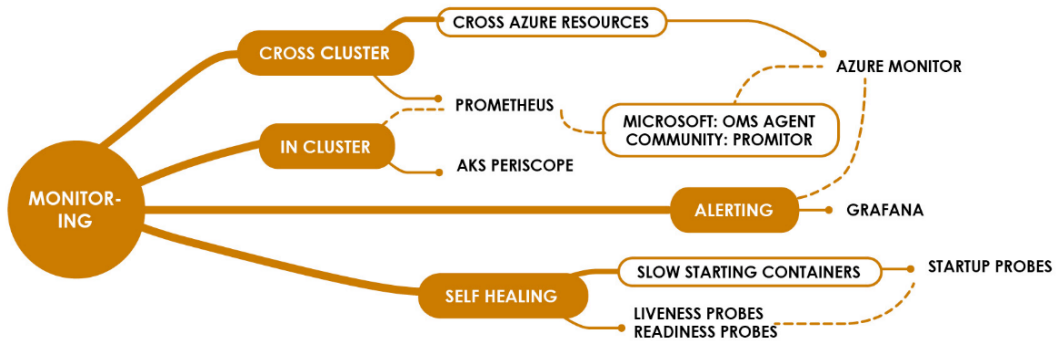


Figure 3.24 – AKS monitoring

Azure Monitor can be used as a single pair of glasses to collect not only Azure logs, but also AKS logs through its **OMS agent daemonset**, which runs on every worker node. Of course, Azure Monitor can do this for one or more clusters. **Prometheus** (<https://prometheus.io/>) is K8s' out-of-the-box monitoring solution. It is often used together with **Grafana** (<https://grafana.com/>) to capture logs and to define dashboards and alerts. Prometheus and Azure Monitor can be used together.

K8s is able to monitor containers, thanks to **liveness and readiness probes**. These will cause the containers to be automatically restarted (by default), in cases in which they do not respond to kubelet's GET HTTP requests (for HTTP workloads). This also allows AKS to restart failed containers with an exponential retry mechanism. This capability is part of the self-healing behavior proposed by container orchestrators. Self-healing goes even further by making sure the actual state is in line with the desired state, defined in the deployment manifest. From time to time, containers might need to store information outside of a database. Let's see which storage options are at our disposal.

Exploring AKS storage options

Storage in AKS can be summarized in one acronym: **CSI (container storage interface)**.

Figure 3.25 displays the different storage options:

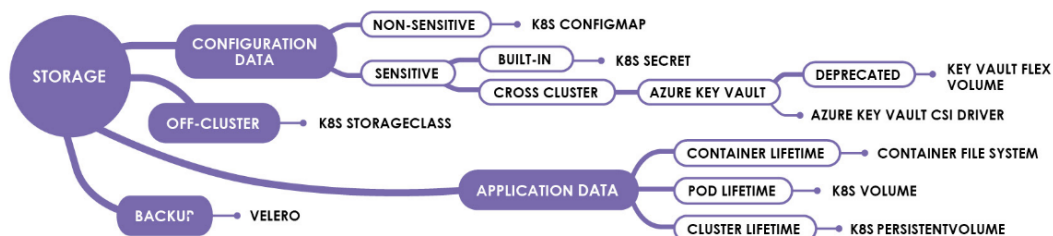


Figure 3.25 – Storage options

Every storage provider can provide storage services through the implementation of CSI. Most of the time, external persistent **volumes** are used to store data. We do not recommend that you store information directly on the container's file system, because of the container volatility. The volume is a way to abstract and store data in a persistent manner, but you should carefully choose the right provider, to avoid performance issues. Configuration data can be transmitted to assets through **ConfigMaps**, while sensitive data should always be persisted in **Azure Key Vault**. This applies to non-AKS workloads as well.

The built-in K8s **secret** is not the most secure way to store sensitive information, as it merely encodes data in base64. That said, secrets are RBAC-protected so not everyone nor every asset can access them, and in Azure, AKS disks are encrypted by default using Microsoft/customer-managed keys. In any case, Azure Key Vault, combined with Azure **Active Directory (AD)** pod identity, should always be the preferred approach.

Now that you have a better idea of how to control the network, deploy and monitor assets, and how to leverage the various storage providers, let's see how to scale your applications.

Scaling AKS

Scaling will be discussed further in the *Reference architecture for microservices* section. However, *Figure 3.26* shows you how to scale both the cluster and the pods. We have already touched upon cluster scaling in the *Zooming in on networking* section:

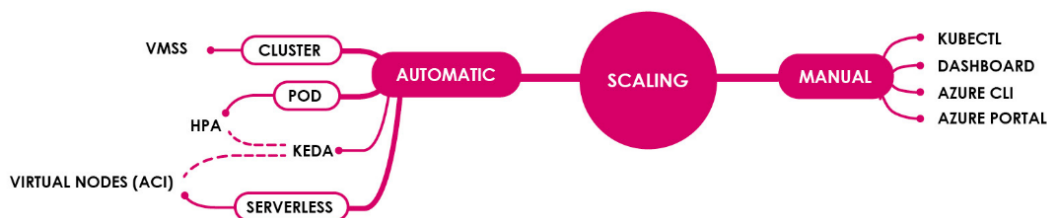


Figure 3.26 – Scaling options for AKS

Horizontal Pod Autoscaler (HPA) is K8's way to scale out pods. It is essentially based on CPU/memory metrics. Metrics providers such as Keda and Prometheus can also feed HPAs to scale pods based on custom metrics, and not only system ones. **Virtual nodes** translate to **ACIs** in Azure and are very useful for bringing extra power to the cluster. Pods can scale automatically, upon a declarative configuration in raw YAML files or any K8s packaging tool, or by using the **Kubectl** command-line tool. Clusters also support autoscaling but manual interventions using the **Azure portal** and the **Azure CLI** are also possible.

We have now looked at the most important AKS topics. Nevertheless, let's go through a few practical aspects when getting started with AKS.

Exploring miscellaneous aspects

As stated earlier, we will not detail every top-level group of this scoped map, for the sake of brevity. Some of the elements depicted in *Figure 3.27* are discussed further in our microservices use case:

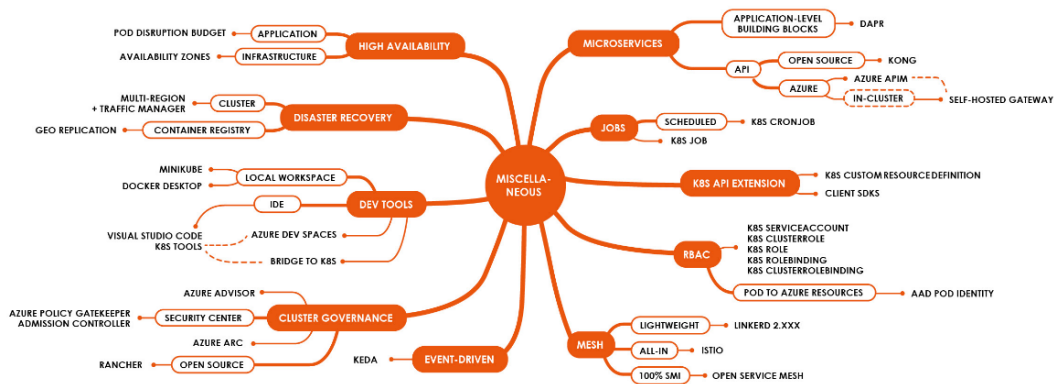


Figure 3.27 – Miscellaneous aspects of AKS

It is interesting to note the **DEV TOOLS** group, which can help developers get started with the tooling they need to begin developing. While **Visual Studio** and **Visual Studio Code** come in handy, it is often not enough to resolve dependencies that a given container might have toward others. It is possible, to a certain extent, to replicate an AKS infrastructure to a local cluster, such as **Minikube** or **Docker Desktop's embedded K8s**, when hosted on the developer machine, but **Azure Dev Spaces** or **Bridge to Kubernetes** may be a better approach for larger applications.

Local replication is a good fit, as long as the total number of containers and dependencies can be handled by the local CPU and memory resources. The value proposal of Azure Dev Spaces is to bridge the IDE to an actual shared AKS for development and testing purposes, which eliminates the need for local replication. It's important to note, however, that Azure Dev Spaces will be retired in October 2023, to the benefit of **Bridge to K8s**, which aims to simplify the development experience. Unlike Azure Dev Spaces, Bridge to K8s is 100% client-side and can be achieved using Visual Studio Code or Visual Studio. So, any greenfield project should start with Bridge to K8s. We keep mentioning Azure Dev Spaces because you might still be confronted with it in the future. Also, it is interesting to note that AKS itself can be extended through the concept of **custom resource definition (CRD)**, should you sell a product or need extra customization for your corporate clusters. Most of the other subgroups are partially discussed in our microservices use case, which comes next. Let's first see why we might want to use AKS instead of native Azure services for microservices architectures.

AKS and service meshes for microservices versus Azure native services

Microservices have become so popular that it is important to understand why AKS is probably the best service to run them in Azure. In a nutshell, the technical promises of microservices architectures are as follows:

- To make a single service the deployment unit and to enable frequent and granular deployments, while not directly impacting the other services.
- To make a single service the scaling unit, to benefit from granular scaling, and to prevent the waste of compute resources.
- To easily segregate mission-critical services from less important ones.
- To have more resilient solutions, since every part is isolated. Unlike a monolith, not everything will crash at the same time.
- To enable multiple teams to work together, at the same time, with different technologies (polyglot).

While all the preceding points could be achieved using Azure App Service and Azure Functions, AKS makes it easier. Consider *Figure 3.28*, which shows what a microservices architecture may look like:

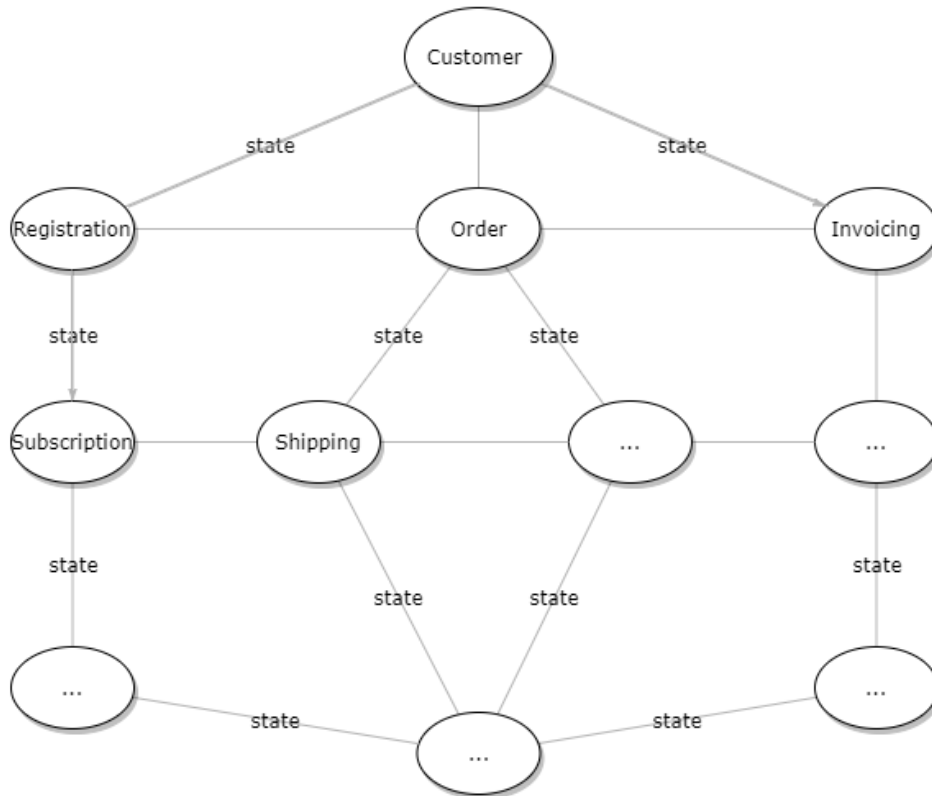


Figure 3.28 – Microservices diagram

We see that many (there could be many more than this) services interact with each other and exchange information through direct and indirect asynchronous calls (through a bus for instance), or they react to events notified by Azure Event Grid. What you see here is literally a **service mesh**. We briefly touched on service mesh products earlier. They typically respond to the preceding requirements, since they offer service-level granularity for whatever concern we might have.

They also ensure a complete overview and secure communication between services. As highlighted earlier, service meshes are layer-7-aware. They know the difference between HTTP/1.1, HTTP/2, and load balance requests accordingly. This might seem anecdotal, but it is not. Many microservices architectures leverage gRPC, which in turn makes use of HTTP/2. Usually, gRPC is the preferred protocol for performance reasons, as it prevents the creation of unnecessary HTTP requests, while optimizing serialization.

As the meshes grow, the extra milliseconds of latency per service call turn into seconds, hence using gRPC or REST over HTTP/2 to mitigate this problem. HTTP/2 reduces the network overhead through its multiplexing feature, which allows clients to reuse the same connection across requests. While this is a great benefit, it is unknown by layer 4 load balancers (including the K8s service component), which do not differentiate HTTP/1 from HTTP/2.

A possible side-effect is that if multiple instances of the same pod are available, an existing client will keep talking to the same instance (should it even be the slowest one), which is not totally aligned with the scale-out story of cloud-native platforms. Service meshes come to the rescue by controlling the HTTP/2 multiplexing feature, according to the actual latency of each available backend instance, which creates the extra HTTP/2 requests when needed, to leverage all the available backend instances. This alone could almost justify the use of a service mesh tool.

Note that K8s alone already natively answers all the requirements of a microservices architecture. A service mesh product will simply improve the overall architecture.

Let's review typical microservices requirements, with Azure App Service and functions versus AKS:

- **Service-level deployment:** Even when sharing the same underlying App Service plan, app services can be deployed independently, since each service has its own `.scm` endpoint. Thanks to deployment slots, Azure App Service can handle canary releases and blue/green deployments. Thanks to Azure App Configuration, it is also possible to enable A/B testing through feature flags. AKS is also able to handle every possible type of deployment.
- **Service-level scaling:** If we want to have full independence between services, we should dedicate an App Service plan to each app service, because per-service scaling is only valid for horizontal, not vertical scaling. While this is technically feasible, it would quickly lead to higher costs since a standard plan is about 60€ (~71\$) a month. In AKS, the pod resource type allows for both independent vertical and horizontal scaling of the service by design, at no extra cost. Function apps on the consumption tier could come to the rescue, but they suffer from cold-start syndrome, so they cannot be used for every scenario. Function apps on prepaid plans lead to the same considerations as App Service.

- **Easy segregation:** With AKS, it is easy to bring special care to mission-critical services. For instance, you can attach **PodDisruptionBudgets** to pods, protecting against voluntary and involuntary disruptions. With App Service and Function apps, all services are treated the same way. Nothing would prevent an operator from restarting an app service by accident.
- **Increased resilience:** Here again, AKS has built-in auto-healing features that keep monitoring at the desired state that you defined against the actual state. Any deviation from the actual state is automatically corrected, providing the errors are transient. App Service also has an auto-heal feature, but auto-healing is slower to kick off speed-wise.
- **To enable multiple teams to work together with different technologies:** Since Azure App Service supports both Linux and Windows containers, technology is not an issue.

Azure App Service and Azure Functions compete quite well, but what they do not have is support for real service mesh software, which helps in building even more secure and resilient systems. Admittedly, service meshes are not always a must-have tool, but they become necessary as your number of services grows, to ensure some coherence and consistency across the entire mesh, which is almost impossible to ensure with a combination of app services and functions. To conclude on this topic, Azure native services will hardly sustain the scaling of a microservice architecture. The next section walks you through a reference architecture for microservices.

AKS reference architecture for microservices – cluster boundaries

Now that we have made a case about the reasons why AKS is a better fit to run microservices architectures than regular Azure App Service and Functions, let's take a closer look at a possible reference architecture. This will happen in two steps: we first focus on the cluster boundaries, then on the cluster internals:

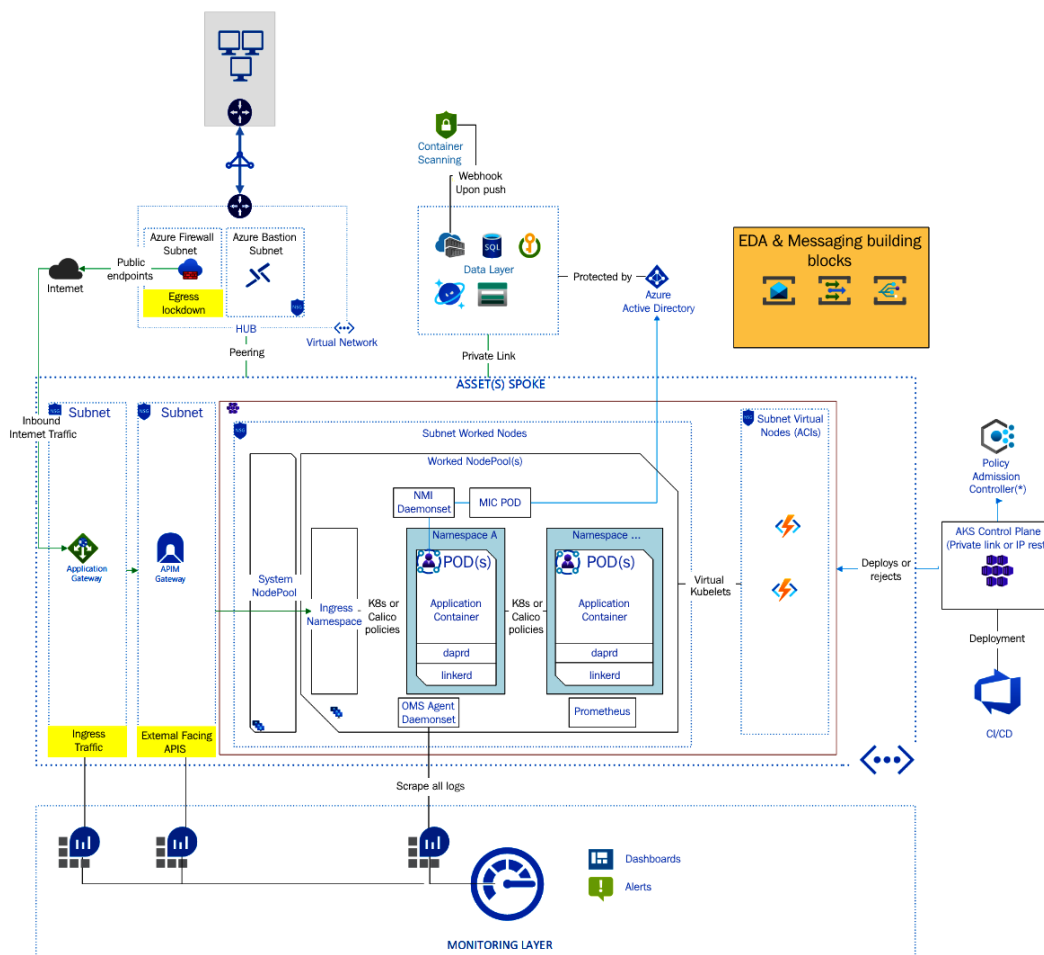


Figure 3.29 – AKS for microservices – cluster boundaries (diagram available on GitHub)

Figure 3.29 maximizes the use of native Azure services. First, we separate the system node pool and the worker node pools. The reason why we split them is to facilitate cluster operations. Here is the rationale behind each choice:

- Splitting node pools:** You may need multiple node pools for scaling reasons (different worker node sizes), but also when working with a mix of concerns. Say, for example, that you have custom AI models, which are served by some of your services. In this case, you might want to train the AI models with specialized node types, while regular services could be scheduled on any node type. You can achieve this using K8 node taints and tolerations.

- **Azure Firewall:** We leverage Azure Firewall for egress lockdown, which we depicted earlier.
- **Azure Application Gateway:** We use Azure Application Gateway as the inbound appliance. It proxies our APIM gateway and is the bridge between the internet and our internal ingress controller. At the time of writing, AFD is only able to connect to publicly accessible backend pools, hence the reason we chose Application Gateway here. Should Front Door be able to connect to private endpoints, we would use it instead of Application Gateway. The main reason to favor Front Door over Application Gateway is its global scale and built-in disaster recovery features.
- **Azure API Management:** The API Management instance is used to expose part of our microservices to the external world. Note that the premium pricing tier can also be fully internal from a connectivity standpoint. We use the Microsoft managed gateway, but there could be good reasons to self-host it in the cluster itself, such as attaching it to a service mesh or combining it with Dapr. Note that since a single instance can have multiple gateways, you might use the Microsoft hosted gateway, plus one or more self-hosted gateways.
- **Azure Monitor:** Although Prometheus is the default monitoring system in the K8s world, it is important to use Azure Monitor, to have a single pair of glasses when looking at the broader picture. Using both Monitor and Prometheus is not mutually exclusive (you can use both), but that comes at higher costs. You might want to filter out what the OMS agent should collect. You can take a closer look at <https://docs.microsoft.com/azure/azure-monitor/insights/container-insights-agent-config> to read more about the OMS agent capabilities.
- **Private link and private cluster:** Depending on your company's DNA, you might rely on a public, private, or restricted API server. The advantage of using the public API server is that you can interact from anywhere. By *interacting* we mean running kubectl commands, having CI/CD pipelines talking to the cluster, and so on. A traditional security architect would actually see this as a disadvantage. You have two other options: IP restrictions, which are just network ACLs that prevent non-whitelisted IP ranges from interacting with the cluster, and a fully private API server. The consequence of such an approach is that CI/CD pipelines may not rely on Microsoft hosted agents anymore. You have to self-host your agents so that they are part of a network perimeter that you control. Note that with network ACLs, you could still use Microsoft-hosted agents by the end of 2020, providing you whitelist the global IP range of the Azure DevOps service tag. This also means that any other Azure DevOps instance would be able to connect to your API server. We use the word **connect**, not **authenticate**.

With private clusters, you also have to rely on Azure private DNS zones to resolve the private endpoint. Note that regardless of the API server connectivity choice, the cluster itself may leverage private-link-enabled data stores to make sure data-related traffic only passes through private connections. There are, however, some major limitations when it comes to EDA building blocks, such as Azure Service Bus. Think twice before enabling private links for these services. You can find more information about these limitations at <https://docs.microsoft.com/azure/service-bus-messaging/private-link-service>.

- **Policy admission controller:** A way to enforce some basic security guidelines on AKS is to use Azure Policy, which extends Gatekeeper v3, an admission controller webhook for **Open Policy Agent (OPA)**. Azure Policy ships by default with some built-in initiatives as shown in *Figure 3.30*:

The screenshot shows the Azure Policy console interface. At the top, there are navigation links for '+ Initiative definition', '+ Policy definition', and a 'Refresh' button. Below this is a filter bar with 'Scope' set to 'Microsoft Azure Sp...', 'Definition type' set to 'Initiative', 'Type' set to 'All types', and 'Category' set to '1 categories'. A search box is also present. The main table lists two initiatives:

Name	Definition location	Policies	Type	Definition type	Category
Kubernetes cluster pod security rest...		8	Built-in	Initiative	Kubernetes
Kubernetes cluster pod security bas...		5	Built-in	Initiative	Kubernetes

Figure 3.30 – K8s initiatives

As an example, you may end up with the policies shown in *Figure 3.31*:

The screenshot shows the 'Policies' tab in the Azure Policy console. There are tabs for 'Policies', 'Assignments (0)', and 'Parameters'. A search box is present with the text 'Filter by reference ID, policy name or ID...'. A button labeled 'All effects' is also visible. Below the search box, a list of policies is shown:

Policy
Do not allow privileged containers in Kubernetes cluster
Kubernetes cluster pods should only use approved host network and port range
Kubernetes cluster containers should not share host process ID or host IPC namespace
Kubernetes cluster containers should only use allowed capabilities
Kubernetes cluster pod hostPath volumes should only use allowed host paths

Figure 3.31 – Sample policies for K8s

This gives peace of mind to your security department, while enabling a shift-left mindset.

- Azure Container Registry:** Any container platform requires a container registry. Here, again, there are many different registry providers available. At the time of writing, private-link-enabled registries cannot leverage Azure Security Center's image vulnerability scanning, which is yet another consequence of using private links. Should you rely on another vendor, make sure to assess the potential extra network latency you might encounter at image pull time, since the registry will not be part of the Azure backbone.

Now that we have seen some important considerations about the cluster boundaries, let's explore the cluster internals.

AKS reference architecture for microservices – cluster internals

With K8s, what happens outside the cluster may be less important than what happens inside of the cluster. Here, we will explain the rationale behind each technology choice we made in *Figure 3.32*, which assumes that we are hosting multiple assets in a shared AKS cluster:

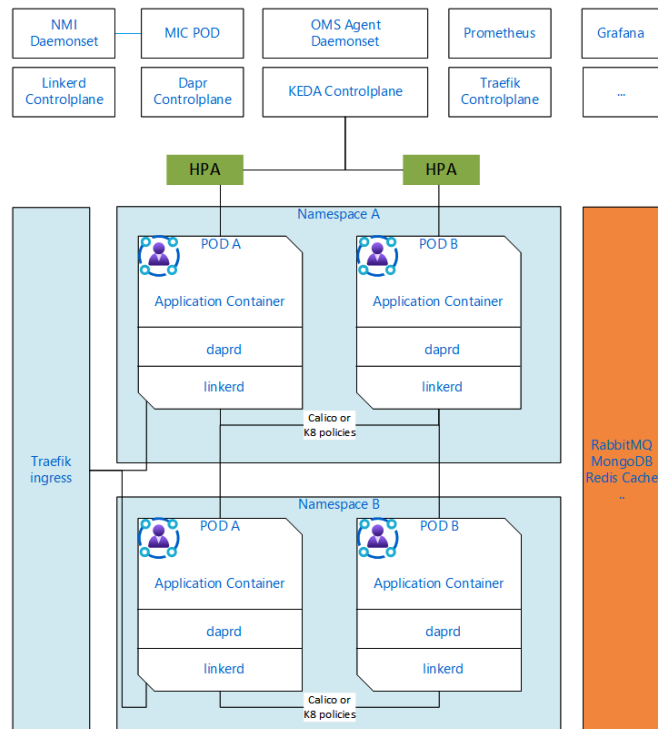


Figure 3.32 – Cluster-scoped view

Let's now review, one by one, each technology mentioned in the diagram:

- Dapr is still a young framework, but it has enormous potential. As we saw in *Chapter 2, Solution Architecture*, Dapr has many connectors to both Microsoft and non-Microsoft event/data stores. Dapr makes it very easy to decouple the application code from the stores it is talking to. You do not need to rely on specific product/vendor SDKs. Do not be confused by *daprd*, mentioned in *Figure 3.32* – it is not misspelled. Daprd is Dapr's sidecar container, which acts as a network proxy for inbound and outbound connections made by your application.
- We have a service mesh for all the reasons explained before (mTLS, better observability, smarter load balancing, and so on). We opted for Linkerd because it is very lightweight and does not incur a big learning curve. Feel free to use another product, should you have specific needs that are not covered by Linkerd. The important thing here is that a service mesh is required in a true microservices architecture.
- We opted for KEDA, to ensure a nice pod/job-level scale-out story. Remember that KEDA is also vendor-agnostic and facilitates interactions with both Azure and non-Azure stores.
- Depending on the cluster networking option (Kubenet or Azure CNI), you might end up with K8s policies, or you might have to use a product, such as Calico, to control network traffic (layer 4) from and to the pods. Calico can also be used with Azure CNI. The goal here is to control the communication between the different bounded contexts, which by default is entirely opened in AKS. This is not AKS-specific; it is just how K8s works. In *Figure 3.29*, we opted for Azure CNI because we want to be able to leverage virtual kubelets through ACIs.
- We opted for the Traefik ingress controller because it is one of the most flexible and dynamically configurable products. Other options could have been Nginx or even **Azure Application Gateway Ingress Controller (Azure AGIC)**, which is currently not at the level of its competitors (feature-wise). However, keep in mind that the cloud is a moving target. Make sure to review AGIC regularly. The benefit of using AGIC over all the others is the delegation of the control plane hosting to the cloud provider.
- We made sure you can enable Azure pod identities, in order to leverage managed identities from within AKS, and to interact with Azure services that support Azure AD authentication.

- On the right-hand side of *Figure 3.32*, you can find some self-hosted products. You should first consider using Azure services instead of self-hosted services. Self-hosting means being fully responsible for high availability, backup/restore, disaster recovery, and so on. Plus, it usually requires worker nodes with a higher memory/CPU profile. For instance, the default MongoDB resource request is 4.5 GB of memory, up to 6 GB, which is already quite a lot for a single container. On the other hand, sometimes self-hosting a component comes with extra benefits. For instance, it is possible to self-host an APIM gateway inside AKS, inject it with the service mesh, and even inject Dapr's `daprd` sidecar, to leverage Dapr-specific APIM policies.

We hope that this two-step architecture approach helped you grasp the important aspects of a microservice architecture, and beyond, of an AKS-based platform. Let's summarize this chapter and see what comes next.

Summary

In this chapter, we did a vertical exploration of infrastructure practice in Azure. We covered several topics, such as networking, monitoring, backup and restore, high availability, and disaster recovery for both Azure itself and AKS. We made it clear that AKS is a special service that comes with its own practices and ecosystem. As an Azure infrastructure architect, you should pay special attention to AKS, whenever it lands on your plate. Our message here is this: yes, use AKS – but do not overlook its complexity and particularities.

In this chapter, we also explored two concrete use cases. The first one demonstrated how challenging (and costly) it can be to have a consistent and coherent disaster recovery strategy for a global API deployment. The second one was about using AKS for microservices. We explained why we think that AKS is more suitable than pure Azure-native services for large microservices implementations. We concluded with a reference architecture for microservices, based on AKS. You should now be better armed to tackle most Azure and AKS infrastructure topics and respond to architecture trade-offs, which are an integral part of an Azure infrastructure architect's job.

Chapter 4, Infrastructure Deployment, will be more hands-on as we will concretely create infrastructure as code templates and deploy them through various channels.

4

Infrastructure Deployment

In *Chapter 3, Infrastructure Design*, we had a 360-degree view of the Azure and AKS infrastructure, but we have not yet seen how to get infrastructure components provisioned in Azure. This chapter focuses on one of the major enablers of the cloud, namely, **Infrastructure as Code (IaC)**. More specifically, we will cover the following topics:

- Introducing **Continuous Integration** and **Continuous Deployment (CI/CD)**
- The Azure deployment map
- Getting started with the Azure CLI, PowerShell, and Azure Cloud Shell
- Diving into ARM templates
- Getting started with Azure Bicep
- Getting started with Terraform
- Zooming in on a reference architecture with Azure DevOps

By the end of this chapter, we expect you to be acquainted with IaC concepts and technologies and you should be able to make the right choices for your own organization and your customers.

Technical requirements

If you want to practice the explanations provided in this chapter, you will need the following:

- **An Azure subscription:** To create your free Azure account, follow the steps explained at <https://azure.microsoft.com/free/>.
- **The Azure CLI, PowerShell, and/or Azure Cloud Shell:** If you want to practice them all, you need access to all three. Otherwise, you can just pick your preferred choice. Azure Cloud Shell is the easiest option because you only need a browser to use it.
- **An Azure DevOps organization:** To create an organization, follow the steps explained at <https://docs.microsoft.com/azure/devops/organizations/accounts/create-organization>.
- **Microsoft Visio:** You will need this to open the diagrams, or use the Visio viewer available at <https://www.microsoft.com/download/details.aspx?id=51188>.

All the code samples and diagrams are available at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/tree/master/Chapter04>.

Let's now get more familiar with the CI/CD concepts.

The CiA videos for this book can be viewed at: <http://bit.ly/3pp9vIH>

Introducing Continuous Integration and Continuous Deployment (CI/CD)

Before diving into CI/CD, let's first step back and reflect on what DevOps means. In most large organizations, the IT department is still divided into siloes. The most common ones are the developers, on the one hand, and the infrastructure teams on the other. You might as well have a separate security team and some middle ground bodies, overseen by a governance body and an enterprise architecture practice. The purpose of DevOps is to act as a bridge between the teams and to break the silo mentality. DevOps is part of a broader digital transformation program that may take years to achieve. The whole point behind digital transformation and DevOps is to gain extra agility and efficiency. However, that's easier said than done!

While the theory is promising, the reality often tends to prove otherwise: resistance of the different teams, misunderstandings on the part of management, a lack of proper skills, people who are out of their comfort zone, a *what's in it for me* attitude, and so on. Ideally, the DevOps practice should be accompanied by a real change-management program. We already discussed the strategic aspects in *Chapter 1, Getting Started as an Azure Architect*.

Back on topic, CI/CD is a practice that is part of a DevOps toolchain. Although CI/CD systems existed before the term DevOps appeared, they were mostly used to manage the application source code life cycle, code build, and code releases. **Infrastructure** was added to the mix, thanks to the native capabilities of the cloud, and that is a game changer! Let's first explore the CI/CD process.

Introducing the CI/CD process

Today, we dare to say that there is no real DevOps practice without CI/CD pushed to its full extent. In other words, one cannot happen without the other. *Figure 4.1* shows a simplified view of the CI/CD process:

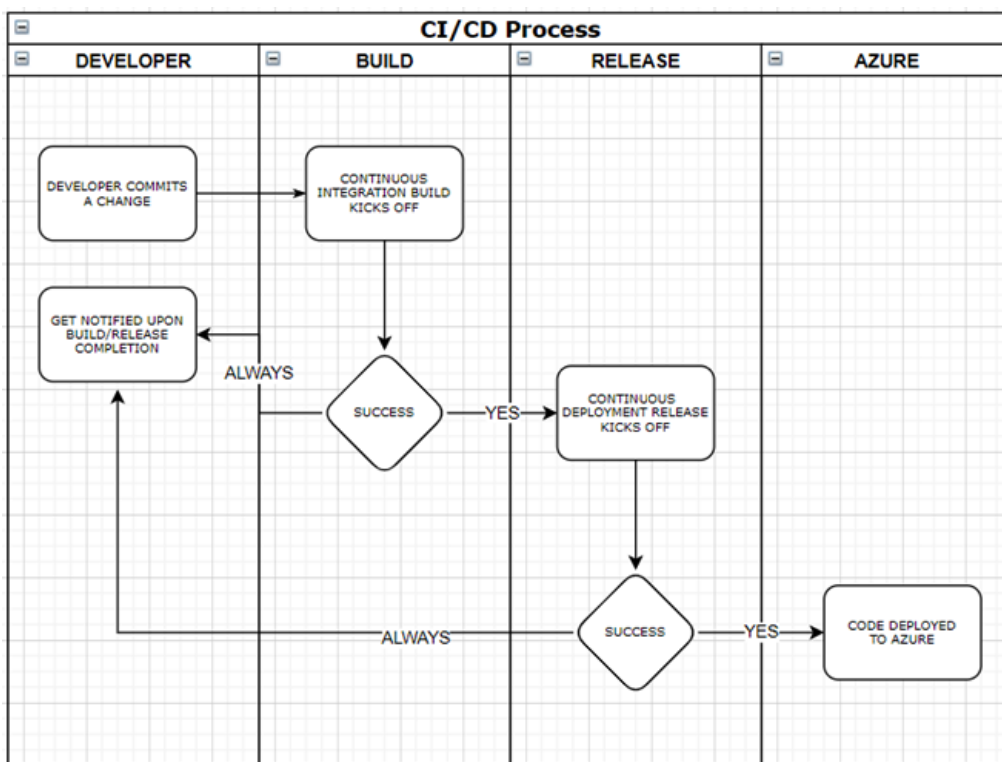


Figure 4.1 – A simplified view of the application CI/CD process

When a developer commits a code change to the code repository, a CI build is triggered. The purpose of this build is to make sure that the change made by the developer does not break the build process. Another important aspect of having a centralized build process is to make sure that all the code dependencies are resolved. Following a successful build, a CD is triggered and the code is deployed to Azure. For instance, it could be deployed to Azure App Service. Let's now see how infrastructure can be added to the mix.

Introducing the IaC CI/CD process

The infrastructure engineer or architect inside of you might think that the aforementioned process is a developer thing. If this came to your mind, then we're sorry to say it, but you might be a little old school. Make no mistake, the same principles apply to IaC. Consider *Figure 4.2* and compare it closely to *Figure 4.1*:

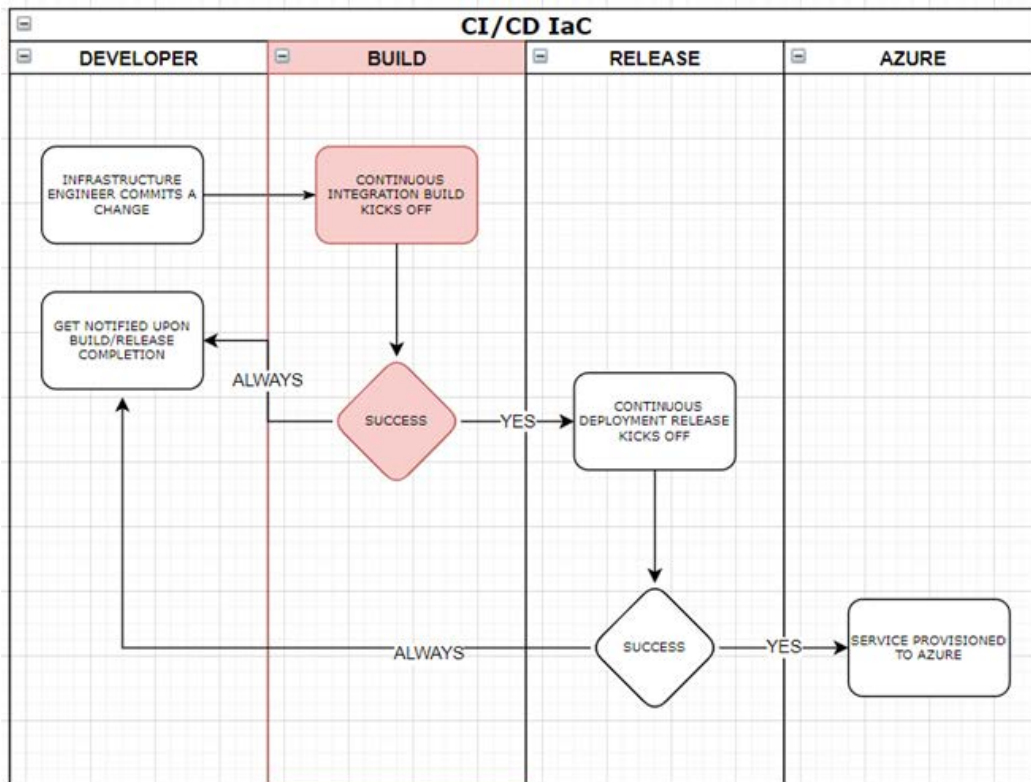


Figure 4.2 – A simplified view of the IaC CI/CD process

There is not much difference between both flows. Of course, just like an application has some source code, so do the infrastructure components. The build part is optional, depending on how you organize your pipelines. As we will see throughout this chapter, the IaC bits are all declarative. A pure IaC template does not need to be compiled, but sometimes it can be packaged or translated into something else to prepare the release. In such a case, the IaC resources will be part of the build artifacts. Whatever you do, you also have to manage pure infrastructure pipelines, and then link them in one way or another to the application pipelines.

The whole point of CI/CD is to automate the entire process, from the developer/infrastructure engineer change, up to the actual object in the target environment. Automation brings you some substantial benefits, including the following:

- Speed of deployment.
- Seamless deployment across environments.
- Standardization of the used services.
- Reliability, since you reduce as many manual steps as possible, which can be error prone.
- Possible disaster recovery mitigation measures. If your level of industrialization is high, it is perfectly sensible to consider redeploying all or part of a solution, in case of a disaster, to another Azure region.
- Continuous releases.
- Going beyond the traditional **DTAP (development – test – acceptance – production)** cycle and quickly spinning up ephemeral environments.
- Better collaboration and control over the assets. You might want to enforce peer reviews and/or approvals prior to releasing anything.
- Improved quality and security, since most tools support quality gates.
- Improved testability.

The list is probably much longer than this! Of course, setting up the factory foundations takes time. In the beginning, setting up a CI/CD pipeline is often perceived as a loss of time, or as an activity that slows down the overall process, and that is correct to some extent. You only realize the aforementioned benefits once you start reusing these infrastructure artifacts across your projects.

The tools, such as Azure DevOps, go way beyond mere technical aspects, as they also allow all stakeholders (functional and technical people) to work together in a consistent and coherent way. Proper tooling and use of that tooling is key to succeeding in your DevOps journey. Let's now take a closer look at our map and analyze the various deployment options.

The Azure deployment map

Unlike the other topics, the Azure deployment map is rather small. We will first elaborate on its different top-level groups, but this time we will be a little more hands-on with a few of our topics. We will compare the different options at our disposal, and then we will show you a real-world example of an advanced IaC implementation with Azure DevOps. We lightly touched on your deployment options in *Chapter 2, Solution Architecture*. *Figure 4.3* shows a more elaborate view of the Azure deployment landscape:

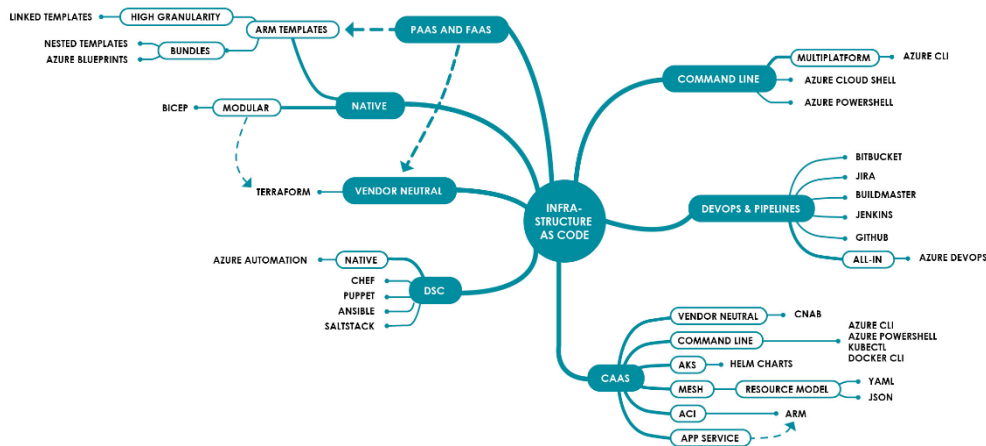


Figure 4.3 – The Azure deployment map

Figure 4.3 has seven top-level groups:

- **DSC (Desired State Control)**
- **PAAS AND FAAS**, which largely encompasses both vendor-neutral and native technologies
- **COMMAND LINE**
- **DEVOPS & PIPELINES**

- **CAAS**
- **VENDOR NEUTRAL** refers to deployment tools and techniques that can be used with other platforms than Azure
- **NATIVE** refers to deployment tools and techniques that can only be used with Azure

DSC is a way to maintain a system aligned with predefined standards. Natively, only **Azure Automation** can be used to enforce DSC against virtual machines. There are numerous third parties that also support DSC, albeit with virtual machines. They all work with virtual machines in any cloud and on-premises, since the only prerequisite is to let the DSC agent contact the control plane, which, in the case of Azure, is reachable at `https://<workspaceId>.agentsvc.azure-automation.net`. One of the biggest advantages of using Azure Automation DSC is to delegate the control plane management to Microsoft, meaning the desired configuration persistence and orchestration. Whether you manage 10, 100, or thousands of virtual machines, the service will scale automatically for you.

For *PaaS and FaaS*, we can rely on Azure **ARM templates**, **Terraform**, and **Azure Bicep**. We will get into the nitty-gritty of this later, but we can already say a few things. ARM templates are Azure's native way to provision infrastructure components. Azure Bicep is the next generation of ARM templates. It is still in its early days, but the goal of Bicep is to simplify the ARM story. You will see later that it can quickly become complex.

Finally, Terraform is another usual suspect. It is vendor neutral. It works with different providers, and Azure is one of them. Most people find Terraform easier than ARM templates, hence the reason Bicep was born. However, Terraform comes with a major drawback: it is often lying behind ARM templates when new services are released in Azure. ARM templates, Bicep, and Terraform all share the fact that they are based on a declarative approach. We will explore these declarative models further in the coming sections.

Unlike a declarative infrastructure, we can always count on *command-line* tools, such as the **Azure CLI**, **PowerShell**, and **Azure Cloud Shell**. PowerShell is well-known, and it has been used by system engineers for many years. It used to be the only imperative approach to provision Azure services. The Azure CLI came later, with the ambition to be cross-platform, from the ground up. Nowadays, the Azure CLI should be preferred over PowerShell because Microsoft invests more in the Azure CLI than PowerShell for Azure. PowerShell is indeed a more generic tool, while the Azure CLI is dedicated to Azure.

Azure Cloud Shell is available from within **Azure portal**, and it can be used with both the Azure CLI and PowerShell. The benefit of using Azure portal is that you only need a browser on your machine to get started. Both the Azure CLI and PowerShell require the command-line tools to be installed on the engineer's machine. Of course, these client tools can also be used from within the CI/CD pipelines. ARM templates do not cover 100% of the Azure resources, which leads us to complement a declarative approach with an imperative one when needed.

In the previous section, we introduced the concept of *DevOps and CI/CD pipelines*, but we have not discussed the tools yet. There are a myriad of tools available. The usual suspects are Azure DevOps, Jenkins, Jira, Bitbucket, BuildMaster, and GitHub. While all these tools can be used to provision Azure resources, we can say from experience that Azure DevOps is probably the best fit. First, Azure DevOps is an all-in-one product. Second, everything is handled by Azure DevOps: the business requirements, source code branching and versioning, advanced build, and release pipelines. All are provided, with pre-defined Azure-related components. On top of all that, the Azure DevOps marketplace counts hundreds, if not thousands, of extensions. But make no mistake, despite its name, Azure DevOps is not only for Azure. You can use the tool to provision to any cloud and even to on-premises environments. Azure DevOps simply has a tighter integration with Azure. Some organizations have a mix of CI/CD tools (Jenkins is almost everywhere), and the good news is that Azure DevOps and Jenkins can work together, so as to let Jenkins build the code and Azure DevOps release it. Jenkins is used in many companies and they often want to keep using it to build all their applications (Azure or not), but use Azure DevOps for the release part, because Azure DevOps has more built-in actionable tasks that help deployment to Azure. Toward the end of this chapter, in the *Zooming in on a reference architecture with Azure DevOps* section, we will see an end-to-end, real-world implementation of IaC with Azure DevOps.

Lastly, in our *CaaS* top-level group, we find **Helm** and raw YAML files for Kubernetes. YAML is simply a generic markup language that has become some sort of JSON superset. The Kubernetes API server understands YAML configurations natively. Helm helps you package and manage Kubernetes applications. ACIs can be provisioned with YAML, ARM templates, and Terraform. Of course, any of the client tools we have just described can be used to deal with CaaS. We must add the **Docker CLI** and **Kubectl** to the mix, to deal with Docker and Kubernetes, respectively. In the next section, we will get more familiar with the client tools.

Getting started with the Azure CLI, PowerShell, and Azure Cloud Shell

In this section, we will give you a glimpse of the Azure CLI and PowerShell from within Azure Cloud Shell. Our goal is not to make you become a scripting rock star, but to just make you familiar with the two approaches. Of course, client tools may be used to provision resources, but they can also interact with Azure in general. Even if you provision everything through CI/CD pipelines, with ARM templates or Terraform, you will still need to retrieve information about the deployed resources. Therefore, we will first focus on getting Azure insights with the client tools in our next section.

Playing with the Azure CLI from within Azure Cloud Shell

As stated before, the Azure CLI should be your default choice when interacting with Azure. If you want to install the Azure CLI locally on your machine, follow the instructions given at <https://docs.microsoft.com/cli/azure/install-azure-cli>. For the sake of simplicity, we will use the Azure CLI from within Azure Cloud Shell, since our goal is just to make you familiar with the Azure CLI, and not to make you write complex scripts. To get started with Azure Cloud Shell, visit <https://portal.azure.com/>. Once in the portal, you will find the Azure Cloud Shell icon, as shown on the right-hand side of *Figure 4.4*:

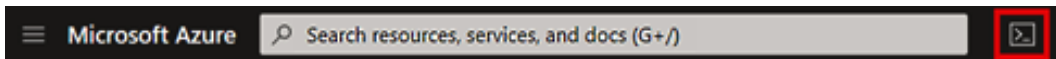


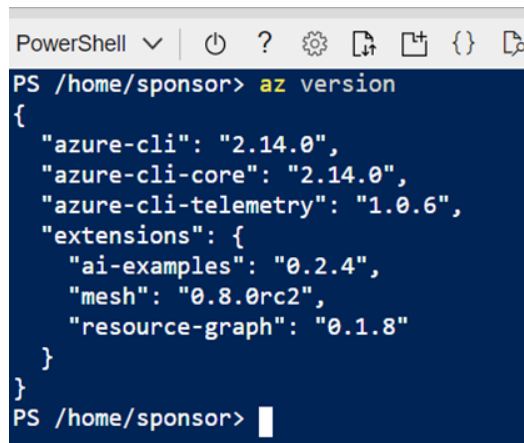
Figure 4.4 – Starting Azure Cloud Shell

Click on the icon. Note that you can access Azure Cloud Shell directly via <https://shell.azure.com>. The first time you use Azure Cloud Shell, it will prompt you to create a storage account for your user. Just accept all the prompts.

Once launched, you can type the following:

```
az version
```

This will show you the current version of the Azure CLI in Azure Cloud Shell:



```
PowerShell | ? ? ? ? ? ? ? ?
PS /home/sponsor> az version
{
  "azure-cli": "2.14.0",
  "azure-cli-core": "2.14.0",
  "azure-cli-telemetry": "1.0.6",
  "extensions": {
    "ai-examples": "0.2.4",
    "mesh": "0.8.0rc2",
    "resource-graph": "0.1.8"
  }
}
PS /home/sponsor> |
```

Figure 4.5 – The az version command in Azure Cloud Shell

As you can see in *Figure 4.5*, some extensions are preloaded. You might have noticed that PowerShell is launched by Azure Cloud Shell. You can switch to Bash if you prefer, but this will make no difference for the Azure CLI itself.

The anatomy of an az CLI command is as follows:

```
az <command group> <parameters> <arguments>
```

For instance, let's say you entered the following:

```
az storage account list
```

This will return all the storage accounts of the current subscription. The preceding command is only a command chain that targets a resource type (storage accounts), but not a resource in particular. To scope it to a specific resource, you can run the following command:

```
az storage account show --name packt --resource-group packtrg
```

Here, we want to view our storage account named packt, which is in the packtrg resource group.

So far, so good. We can go a little further and extract specific information from that storage account. Let's get its keys, using the following command:

```
az storage account keys list --account-name packt --resource-group packtrg
```

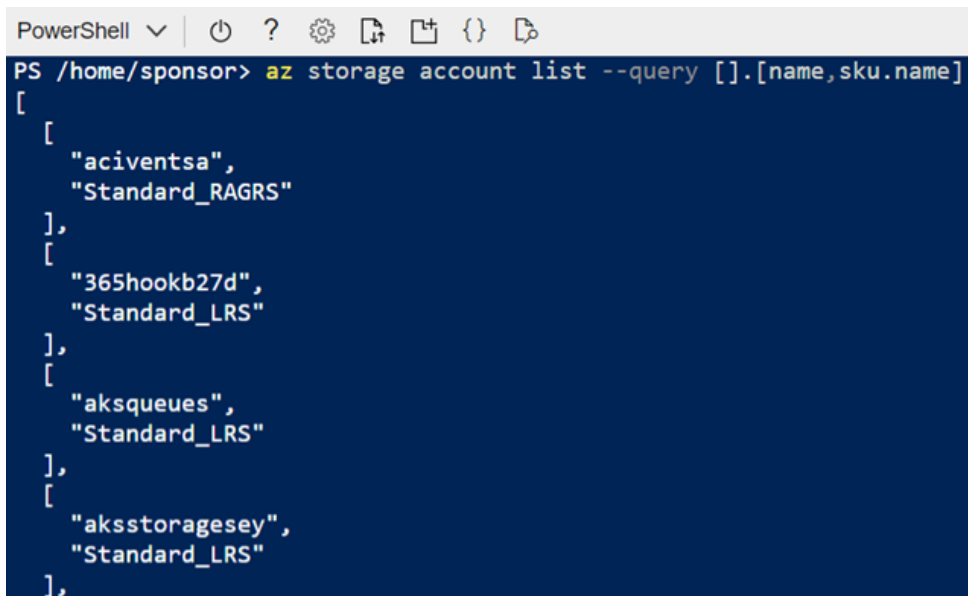
Note the inconsistency between the previous command and this one. The `-name` flag got transformed into `-account-name`. The depth of the command chain really depends on the resources that you work with. Every Azure CLI command supports the following general flags:

- `--help`: To get extra help and examples of use
- `--output`: To tell the Azure CLI how the output should be rendered
- `--query`: To do advanced queries against resources, using the JMESPath syntax
- `--verbose`: To get the most details regarding a given command execution
- `--debug`: To get even more details regarding a given command execution

The Azure CLI is quite straightforward to learn. The only slightly more complex flag is the `-query` flag. Let's start with the following simple query:

```
az storage account list --query [].[name,sku.name]
```

The preceding query returns an array of storage accounts and `sku.name`, as shown in *Figure 4.6*:



```
PowerShell | ? | ? | ? | ? | ? | ?
PS /home/sponsor> az storage account list --query [].[name,sku.name]
[
  [
    "aciventsa",
    "Standard_RAGRS"
  ],
  [
    "365hookb27d",
    "Standard_LRS"
  ],
  [
    "aksqueues",
    "Standard_LRS"
  ],
  [
    "aksstoragesey",
    "Standard_LRS"
  ],
]
```

Figure 4.6 – A list of storage account names and SKU

You may have noticed the composed construct `sku.name`, which enables you to extract a single property of a sub-object of the storage account. If you are unsure about the names and properties to use, then you can always first check the structure of a resource, in this case, a storage account. See the following command:

```
az storage account show --name packt --resource-group packtrg
```

Figure 4.7 shows a truncated response:

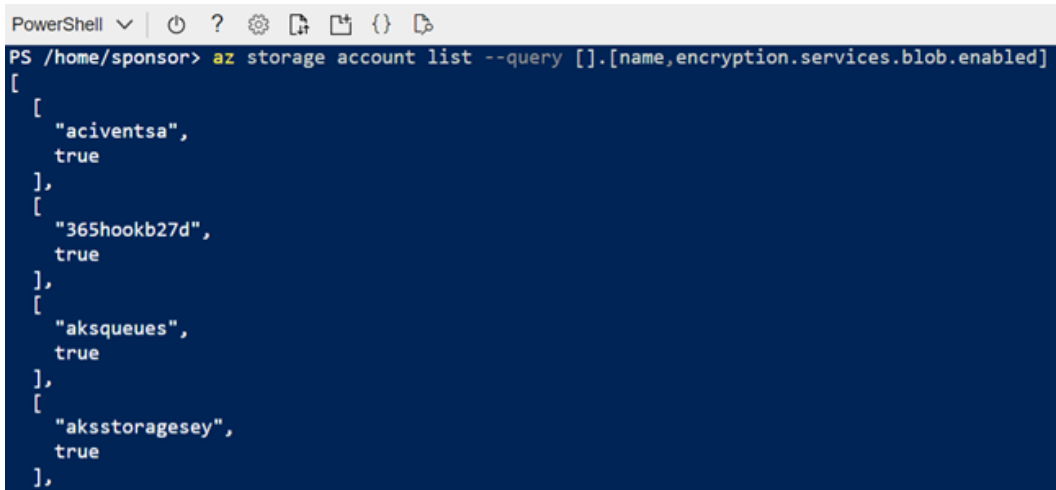
```
{
  "accessTier": "Hot",
  "allowBlobPublicAccess": null,
  "azureFilesIdentityBasedAuthentication": null,
  "blobRestoreStatus": null,
  "creationTime": "2020-01-17T17:44:50.300055+00:00",
  "customDomain": null,
  "enableHttpsTrafficOnly": true,
  "encryption": {
    "keySource": "Microsoft.Storage",
    "keyVaultProperties": null,
    "requireInfrastructureEncryption": null,
    "services": {
      "blob": {
        "enabled": true,
        "keyType": "Account",
        "lastEnabledTime": "2020-01-17T17:44:50.362540+00:00"
      },
      "file": {
        "enabled": true,
        "keyType": "Account",
        "lastEnabledTime": "2020-01-17T17:44:50.362540+00:00"
      },
      "queue": null,
      "table": null
    }
  },
}
```

Figure 4.7 – A storage account truncated structure

With the output of *Figure 4.7*, we can see that storage accounts have an encryption property that is a complex object, but we can easily infer a query argument from the preceding screenshot by identifying the path of a given property. For instance, if we want to view all storage accounts and whether the blob service is enabled or not, we can run the following command:

```
az storage account list --query [].[name,encryption.services.blob.enabled]
```

We get the following response:



```
PowerShell | PS /home/sponsor> az storage account list --query [].[name,encryption.services.blob.enabled]
[
  [
    "aciventsa",
    true
  ],
  [
    "365hookb27d",
    true
  ],
  [
    "aksqueues",
    true
  ],
  [
    "aksstoragesey",
    true
  ],
]
```

Figure 4.8 – The storage account names and blob service status

However, the `--query` flag is not only about selecting properties; it is also about filtering. For instance, the following command returns all the storage accounts that have either a blob or file service enabled:

```
az storage account list --query "[?encryption.services.blob.enabled || encryption.services.file.enabled].{Name:name,File:encryption.services.file.enabled,Blob:encryption.services.blob.enabled}"
```

So far, we only had empty brackets, `[]`. This is the placeholder to specify any query. The second part is about specifying the properties to be shown. The Azure CLI supports logical operators, such as AND (`&&`), OR (`| |`), and NOT (`!`), as well as typical comparison operators, such as `<`, `>`, and `<=`. For complete information regarding the JMESPath language, you can visit their website at <https://jmespath.org/>.

Let's now perform similar manipulations with PowerShell instead of the Azure CLI.

Using PowerShell from within Azure Cloud Shell

The good news with Azure Cloud Shell is that you can work seamlessly with PowerShell and the Azure CLI. You might already be familiar with PowerShell, as it is commonly used for non-Azure-related tasks. The Azure PowerShell cmdlets were available long before the arrival of the Azure CLI. That is why you will still see a lot of Azure-related PowerShell scripts in the coming years. As with the Azure CLI, you can simply launch a new Cloud Shell window, which defaults to PowerShell, as illustrated by *Figure 4.9*:

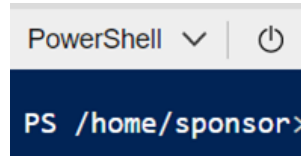


Figure 4.9 – Azure Cloud Shell defaults to PowerShell

The first thing to note is that there are many PowerShell modules for Azure. The latest cmdlets to use come from the **PowerShell Az** module. They supersede the former **PowerShell AzureRM** module, although probably millions of scripts are still running AzureRM in production today. Since we strive to look to the future, we will only focus on PowerShell Az. To see which PowerShell version Azure Cloud Shell is currently running, you can type the following command:

```
Get-Host | Select-Object Version
```

That should return at least version 7.0.3:

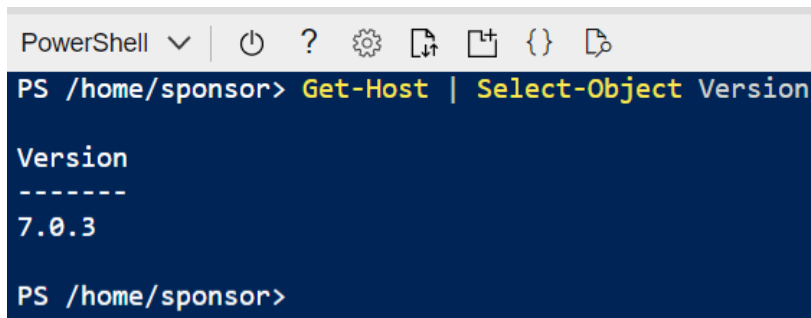


Figure 4.10 – Getting Azure Cloud Shell's current PowerShell version

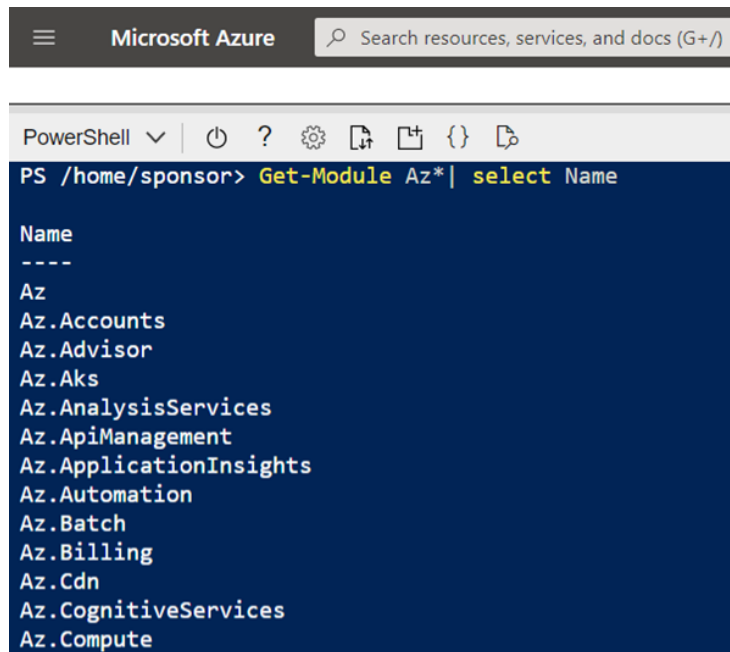
The anatomy of a PowerShell command is as follows:

```
<command> <optional parameters> <optional pipes>
```

PowerShell makes intensive use of the pipeline. In other words, it is easy to pass the output of a first command to the input of the second, and so on. Unlike the Azure CLI, which is specifically designed for Azure, PowerShell includes many more commands. To see the commands from the Az module, you can either run `Get-Command` with all sorts of wildcards, or use the following command to list the modules available:

```
Get-Module Az* | select Name
```

Figure 4.11 shows a truncated list of Az modules:



```
Microsoft Azure Search resources, services, and docs (G+/)
PowerShell | ? ? ? ? ? ? ? ?
PS /home/sponsor> Get-Module Az* | select Name
Name
----
Az
Az.Accounts
Az.Advisor
Az.Aks
Az.AnalysisServices
Az.ApiManagement
Az.ApplicationInsights
Az.Automation
Az.Batch
Az.Billing
Az.Cdn
Az.CognitiveServices
Az.Compute
```

Figure 4.11 – A truncated list of Az modules

You can refine the list of available cmdlets by scoping to a specific Azure resource type. For example, to get the list of Azure Storage-related commands, use the following command:

```
Get-Command -Module Az.Storage
```


Understanding the one that rules them all

Before jumping into our declarative deployment options, it is important to first step back and understand what is the single endpoint that rules everything.

No matter which tool or language you choose, they will ultimately all talk to the same Azure API, namely, the **Azure Resource Manager (ARM)** endpoint. The commercial endpoint is `https://management.azure.com/`. Any call to this endpoint requires the caller to provide a valid **access token**, retrieved from **Azure Active Directory**. Remember, in our *Understanding the ARM template deployment scopes* section, we discussed the least privilege approach and the empowerment of the deployment tools. That is what this access token will be validated against, for any interaction with the ARM API.

Terraform, Azure Bicep, native ARM templates, and imperative client tools all talk to the ARM endpoint. This is important, because if a feature is not exposed through the ARM endpoint, none of the tools will be able to overcome this. *Figure 4.13* shows the underlying plumbing of ARM:

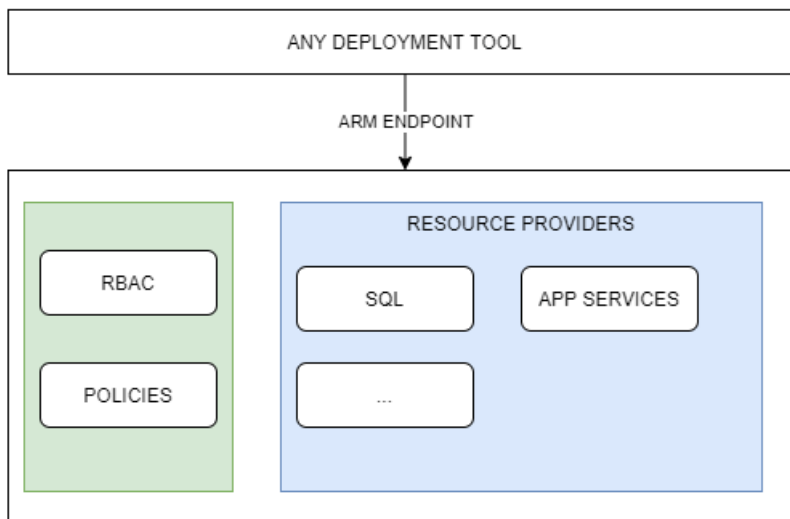


Figure 4.13 – ARM's underlying plumbing

A way to quickly grasp how this API works is to proxy the traffic between the Azure portal and the management endpoint with Fiddler (or any similar tool). You can go to the Azure portal and start Fiddler with a filter (to reduce noise), as shown in *Figure 4.14*:

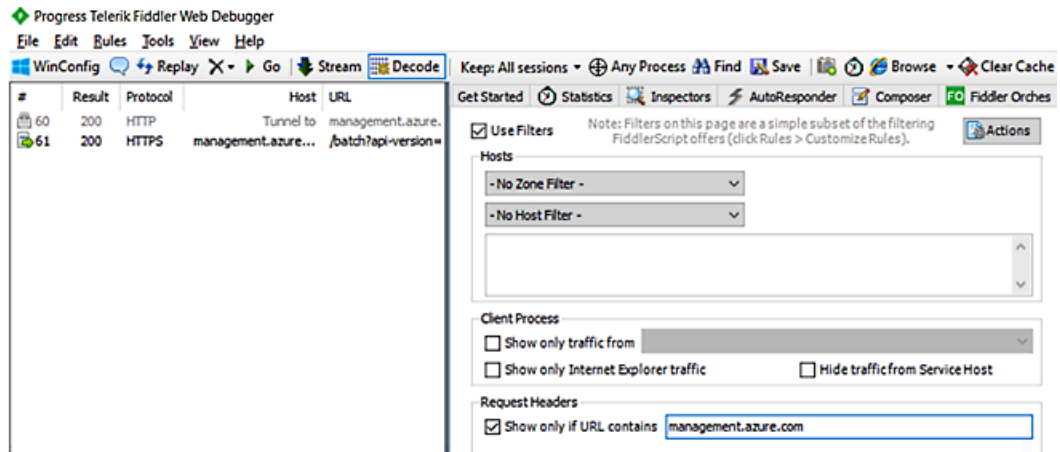


Figure 4.14 – Starting Fiddler with a filter

If you click around in the portal, the requests will be captured by Fiddler. After inspecting these requests, you can easily find out how the ARM API works. In *Figure 4.15*, we can also see the access token that was requested by the Azure portal to interact with the ARM endpoint, on our behalf:

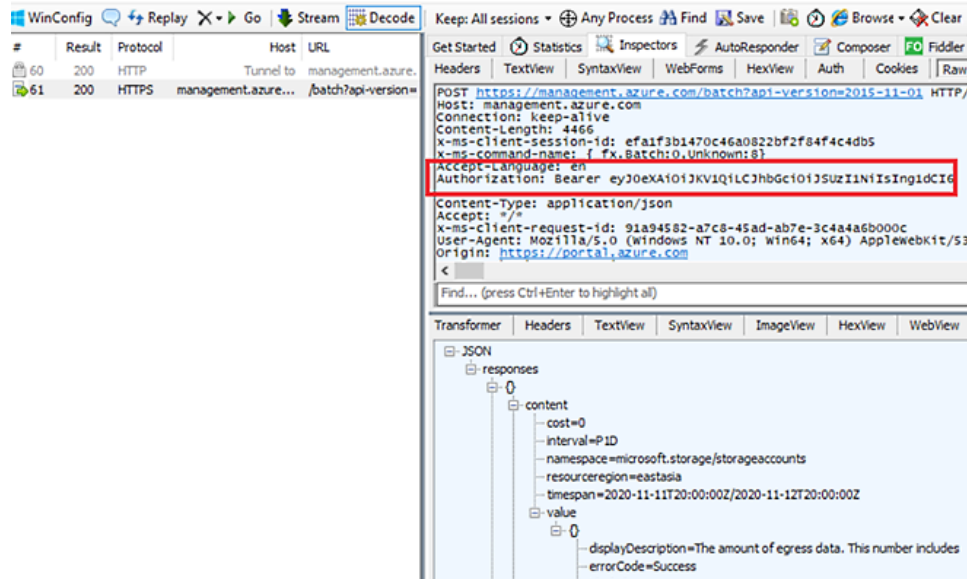


Figure 4.15 – The access token and ARM interactions that are captured by Fiddler

From time to time, you might need to interact with the ARM endpoint yourself, directly from a PowerShell script or a custom application. It is interesting to understand how it works, under the hood.

Now that we have clarified how every tool interacts with Azure, let's explore one of them, namely, ARM templates.

Diving into ARM templates

ARM templates are Azure's native way of provisioning resources in Azure. Almost everything can be deployed through ARM templates, although they do not cover all the Azure services. In this section, we will dive into the ARM world and will provision some services, so as to have a more hands-on experience. Let's get started!

Getting started with ARM

In the real world, it is unlikely that you will ever build an ARM template from scratch. To get you started with ARM, here are a few important handy sources and tools:

- ARM Quickstart Templates is a repository of about 950 templates, and is available here: <https://azure.microsoft.com/resources/templates/>.
- The export wizard of the Azure portal. You can create an Azure resource with the portal and export the template afterward. While this method can be interesting in some situations, where the documentation is a little unclear, exported templates cannot be reused *as-is* to provision other resources.
- The Visual Studio Code ARM extension helps you author templates.

Let's now see the deployment methods of an ARM template.

Understanding the ARM template deployment methods

ARM templates have three deployment methods:

- **Single template files**
- **Nested template files**
- **Linked templates**

Single and nested templates both rely on a single file. This means that all the resources used by a given solution are declared and configured in that single file. This is easier to deploy to Azure, because you have a single file to send, but it suffers from a major drawback: a lack of standardization. Because, when working with a single file, you will have to manipulate it over and over for each project, which is sub-optimal, in terms of reusability. Nested templates were an attempt to maximize reusability with single files, but, in most mature environments, only linked templates are used.

Linked templates allow you to define per-service templates and assemble them into a master file, which is specific to a given application. As shown by *Figure 4.16*, only the master file varies from one project to another, while the shared templates remain the same. Even better, these shared templates can be versioned in a way that's similar to a shared code library:

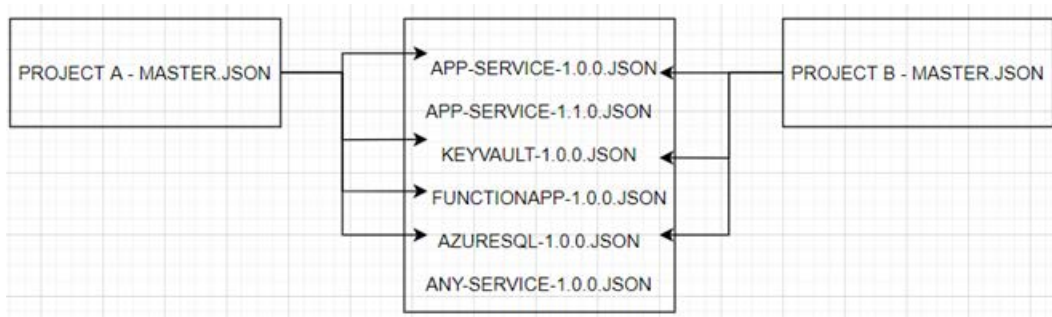


Figure 4.16 – Shared linked templates across projects

Therefore, shared templates are standardized and reused across assets, which is really what you are looking for when building an automation factory. Let's now see the scopes to which templates can be deployed.

Understanding the ARM template deployment scopes

Templates (whether single or linked) are always deployed against an Azure scope. These scopes are as follows:

- **Management groups**
- **Subscriptions**
- **Resource groups**

Consider the following hierarchy, as shown in *Figure 4.17*:

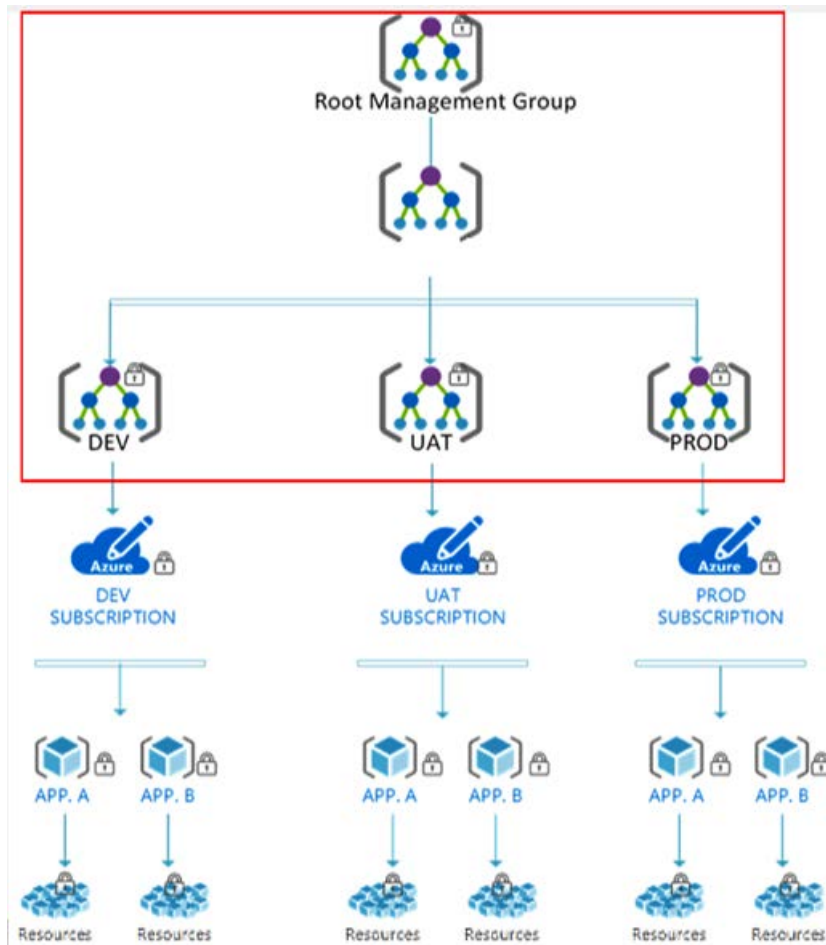


Figure 4.17 – ARM template scopes – Management groups

The **management groups**, highlighted in *Figure 4.17*, are one of the deployment scopes. Typically, templates targeting management groups relate to Azure policies and Azure **RBAC (role-based access control)**. As we have seen in *Chapter 2, Solution Architecture*, Azure policies allow you to build actionable governance, and they enforce a set of controls against resources on the verge of being deployed, as well as against already deployed resources. RBAC allows you to define who/what has access to a given resource, which in this case is the management group. Both RBAC role assignments and policies can be configured through ARM templates.

Next, *Figure 4.18* shows you the **subscription** scope:

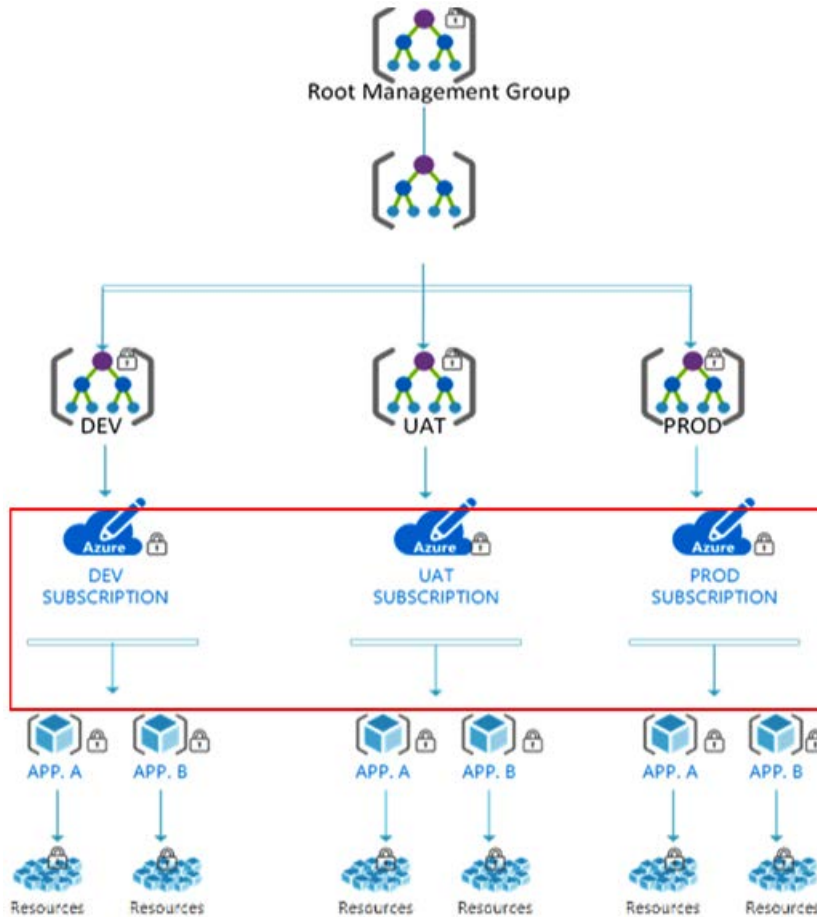


Figure 4.18 – ARM template scopes – Subscriptions

A management group may contain one or more subscriptions (typically more than one). For example, you might have a DEV management group that has more flexible policies and RBAC than the UAT and PROD groups. But you can also have different policies and RBAC configurations per subscription, which can also be deployed through ARM templates.

On top of RBAC and policies, ARM templates can also provision resource groups themselves, when targeting the subscription scope. To do so, the factory must have a subscription-level contributor permission, which is almost never the case in production, except in very mature and cloud-native organizations. Therefore, in non-mature organizations (probably 99.99% of them), resource groups are deployed manually. A service principal, with contributor permission over the resource group, is provided to the people in charge of setting up the CI/CD pipelines. That pipeline makes use of that **SPN (Service Principal Name)** to provision resources to the resource group, which is our next scope.

The last scope is the **resource group**, as shown in *Figure 4.19*:

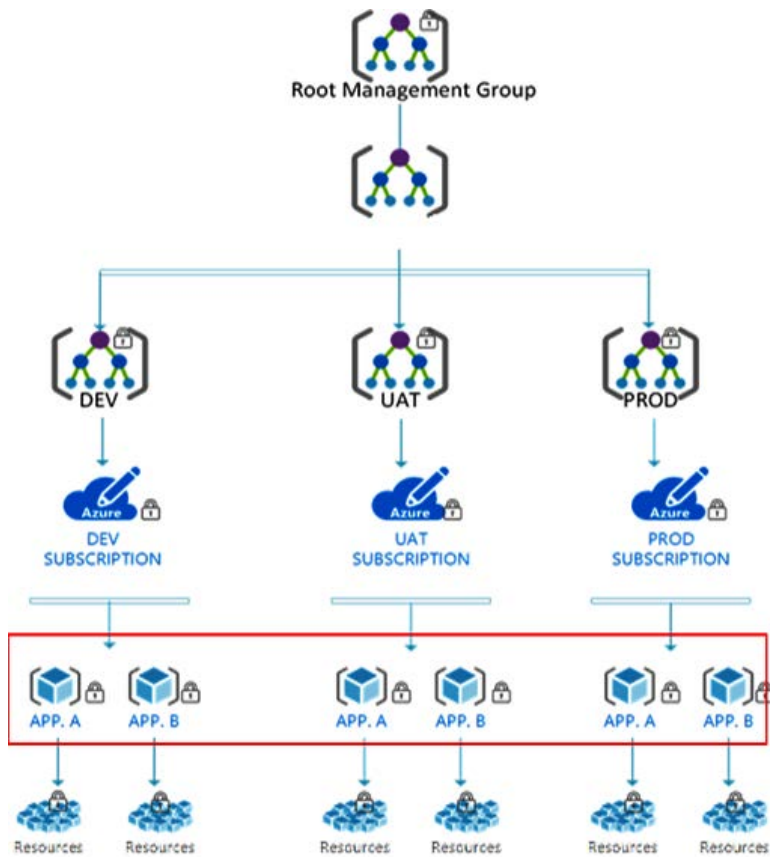


Figure 4.19 – ARM template scopes – Resource groups

Resource groups usually contain the resources for a single project. You may also have shared resource groups (for example, hosting a SQL elastic pool or any other shared service for cost optimization purposes). Other than this, every project-specific resource will be hosted in the project's dedicated resource group. To do so, you are required to have the resource group contributor permission level, and you are also required to have the owner permission level, if you also deploy RBAC and policies.

The scope hierarchy is important because inheritance is applied by default. A policy or role that is applied against a management group will cascade down to all its children (sub management groups, subscriptions, resource groups, and resources). Therefore, it is important to thoroughly think through your factory, in order to define the proper permission levels for the CI/CD pipelines (while keeping the least privilege approach in mind). That is where the maturity of the organization (in the areas of cloud practices) plays an important role. A lack of maturity usually leads to a lack of trust in factory tools and a lack of trust in Azure.

Beware that a single manual step alone, in the entire automated process, can ruin most automation efforts. A rule of thumb is that the more you automate, the more you need to empower tools, and ultimately, the more you need to trust them. Setting up the factory is a part of your cloud foundation, but it is also quite disruptive vis-à-vis traditional IT practices. However, if your company has genuine cloud ambitions, it is important to invest in a well-designed and empowered factory to get the cloud promises realized. In our *Zooming in on a reference architecture with Azure DevOps* section, we will show you how to set it up, but let's first explore the deployment modes.

Understanding the ARM template deployment modes

ARM templates support two deployment modes:

- **Complete**
- **Incremental**

For the sake of brevity, we will quickly review both modes, but you should note that only the incremental mode is supported with linked templates. So, it is unlikely that you would ever use the complete mode.

Complete mode, as its name indicates, will make sure that the outcome of a deployment will match exactly what is defined in the template. To understand this better, let's go through a scenario:

Application A requires an app service, a key vault, and a SQL database. You have defined a single template, with the three resource types, and you used the complete deployment mode. Application A is now deployed to its resource group, which currently contains all the necessary resources.

Later, the application owners realize that they forgot to request an application insights component. For the sake of time, and because it is urgent, you quickly provision the application insights component via the Azure portal. Application A now has four components. Next, Application A's code is modified, and you redeploy it through your CI/CD pipelines. Since ARM templates are idempotent, you redeploy them too, but you did not include the application insights component (you forgot that it was manually provisioned). Because you opted for the complete mode, the application insights component gets deleted by the resource manager.

With incremental templates, the application insights components would remain because they only add or modify the services that are referenced in the template. They do not touch any other resource that would have been provisioned via another channel.

Whether you use the complete or incremental mode, a rule of thumb is to only provision through CI/CD to prevent any accidental resource removal as just explained, which is never pleasant. It's interesting to note that ARM templates are idempotent (meaning that they can be applied multiple times), so it will not break, even if you redeploy the same template 10 times against existing resources. However, if a deployment fails in the middle, there is no real rollback mechanism that would reset your changes. Instead, you can either redeploy until the failure (which may be transient) is fixed, or you can tell ARM to redeploy the last successful deployment. So, instead of a real rollback, it overwrites the current changes with the last successful deployment.

Now that we have covered the high-level functionalities of ARM templates, let's look at the template's internals.

Understanding the anatomy of an ARM template

Figure 4.20 shows you the most frequently used sections that are part of ARM templates:

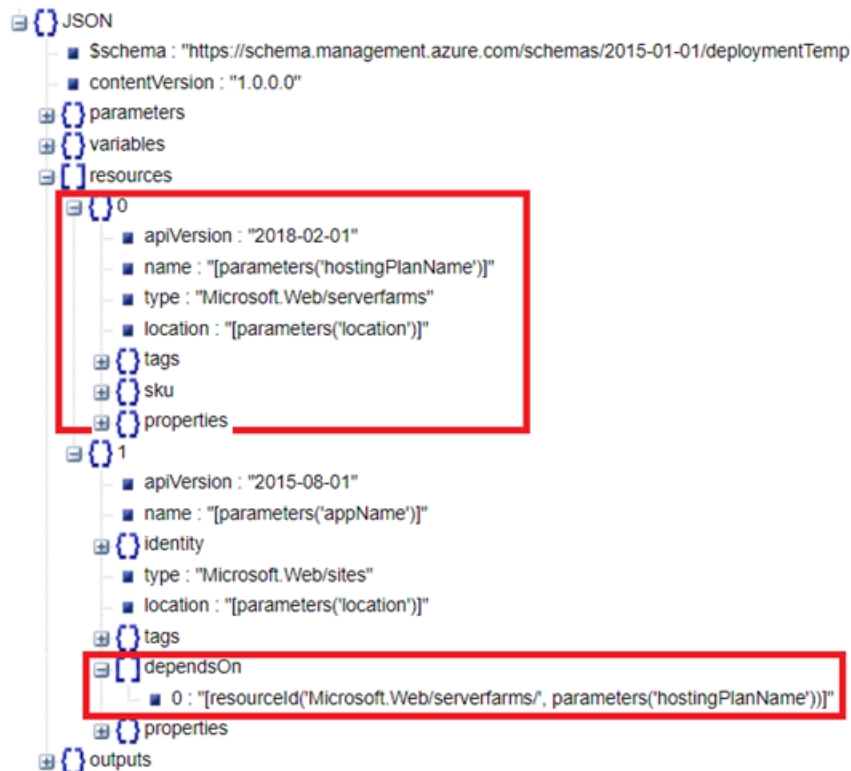


Figure 4.20 – ARM templates

Let's have a closer look at the sections shown in the preceding diagram:

- The **parameters** section makes templates more dynamic. Remember that we are aiming to reuse templates across projects. Parameters are a way to inject dynamic values during the deployments.

- **Variables** are local to the template and allow you to define, once and for all, values that are reused everywhere within the template. An example of this could be the concatenation of two parameters.
- The **resources** section is, of course, the most important one. This is where you define all the resources that must be provisioned by the template. In our previous example, that section would have contained the app service, the SQL server instance, and the key vault.
- **Outputs** makes it easy to define variables that can be reused outside of the ARM template itself. For example, if you provision a storage account, you might output its keys so that they can be reused elsewhere during the deployment.
- The **functions** section lets you define custom functions.

Besides the custom functions, the ARM language exposes many built-in functions, as follows:

- **Array and object functions:** `last()`, `skip()`, `min()`, `max()`, and many others.
- **Comparison functions:** `equals()`, `greater()`, and so on.
- **Deployment functions:** `variables()`, `parameters()`, and others.
- **Logical functions:** `and()`, `or()`, and so on.
- **Numeric functions:** `int()`, `float()`, `mod()`, and more.
- **Resource functions:** `listKeys()`, `resourceId()`, and several other functions. Resource functions are used heavily.
- **String functions:** `concat()`, `indexOf()`, `startsWith()`, and suchlike.

Mastering these functions is important, especially the resource-related functions.

Resources might depend on each other. For instance, *Figure 4.21* shows a template that deploys both an app service and an app service plan:

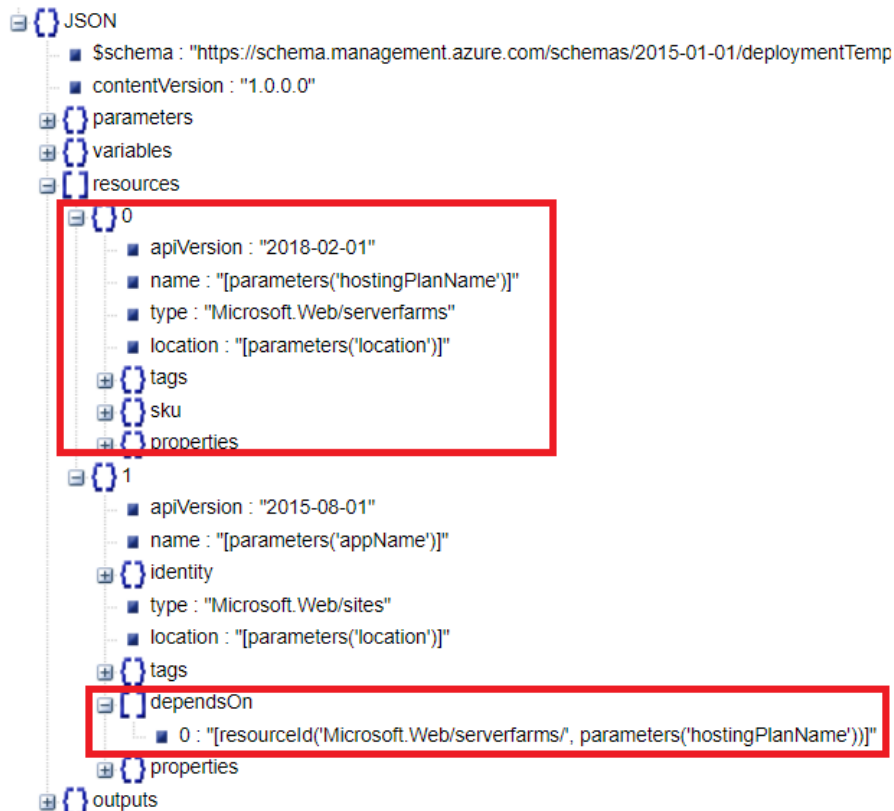


Figure 4.21 – ARM template dependencies

The highlighted parts of *Figure 4.21* show the dependencies in action. The second resource depends on the first one and declares this dependency explicitly. You might have noticed that we use the `resourceId()` function to get the identifier of the app service plan.

The reason why we must take dependencies into account is because Azure's resource manager API will always try to deploy all the resources of the template in parallel, so as to optimize your deployment speed. However, in the preceding example, we cannot deploy the app service if we have not yet deployed its underlying compute building block (namely, the app service plan). Microsoft could have handled the dependencies for us, but unfortunately, we must manage that ourselves. That may seem trivial, but it can quickly become challenging in large applications, or when working with shared services. Fortunately, Azure Bicep (refer to the *Getting started with Azure Bicep* section) helps in this regard.

When using single template files, you usually declare all the dependencies across your resources, in that template. When using linked templates, you usually declare dependencies across your deployments, directly in the master file, as we will see in the next section.

Building a concrete example using linked templates

To gain a better understanding of how it works, let's go through a very basic scenario, which should make you understand the most important aspects of using linked templates. The template files are available in the GitHub repo at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/tree/master/Chapter04/IaC>.

Figure 4.22 shows how the master file interacts with our linked templates:

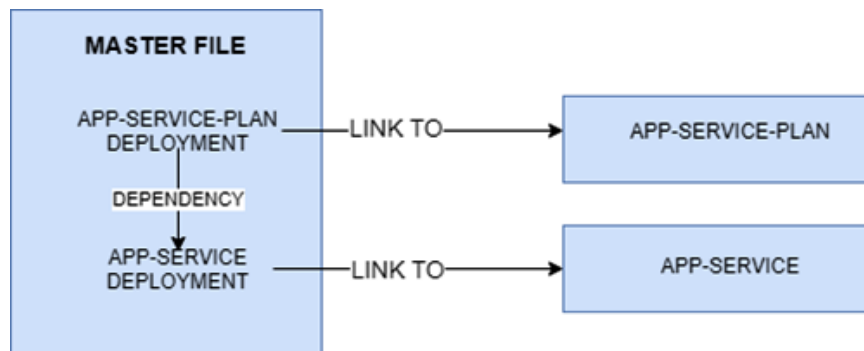


Figure 4.22 – An app service and its plan that uses linked templates

For the sake of simplicity, we will perform the following manual steps:

1. Create a resource group in Azure. Just go to your subscription and create a resource group named `packt`.
2. Create a storage account, and create a container named `templates`, with anonymous access enabled.
3. Upload both the `app-service-plan.json` and `app-service.json` files into the container.

Ultimately, your environment should look like *Figure 4.23*:

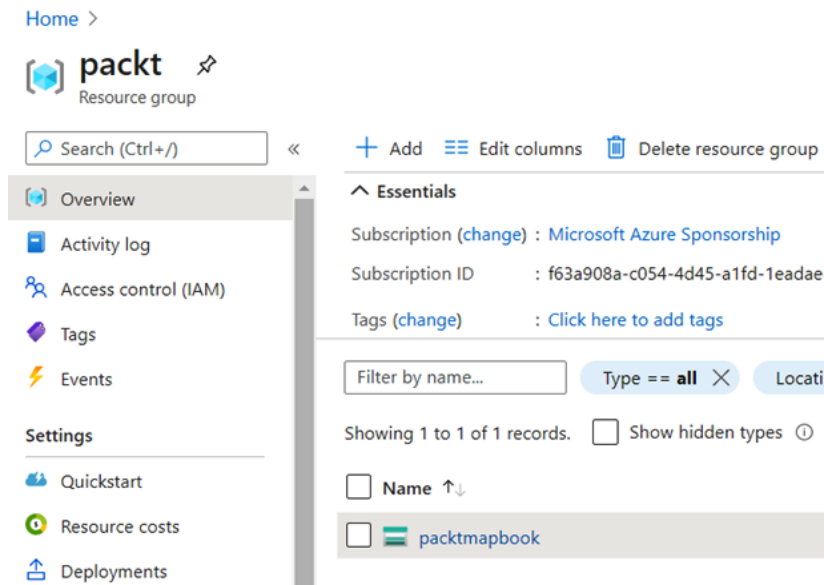


Figure 4.23 – The resource group and its storage account

Figure 4.24 shows you the two template files that are stored in the `templates` container of the storage account:

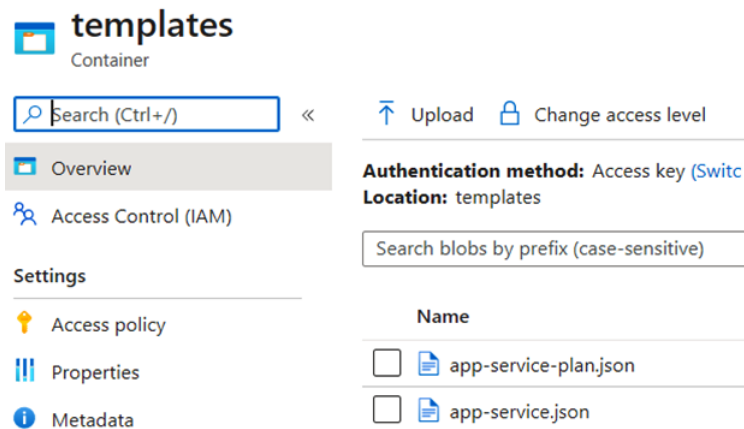


Figure 4.24 – The templates files within the container

Note that you can skip these manual steps and go straight to the explanations if you do not want to follow along with this little exercise.

In the real world, the storage account we use would be secured by using shared access signatures. We used anonymous containers to make our life easy. The reason why a storage account is needed is because during deployment, we only send the `master.json` file to the ARM endpoint. Then, ARM needs to be able to download the linked templates from a connectable (and ideally secure) store, hence, the storage account. It is possible to use any other publicly available storage system that allows you to pass a key, or some sort of authentication mechanism, through the URL.

Let's now focus on the contents of the templates. We'll start with `app-service-plan.json`:



Figure 4.25 – An app service plan template

Figure 4.25 starts with a few **parameters**. Every parameter (that has a default value) is optional. We basically want to know the location, name, pricing tier, and the number of instances that should be provisioned for this app service plan. Next, we have the **resources** section.

We have a single resource of type **Microsoft.Web/serverFarms** that is configured with our parameters. We finish with the **outputs** section, where we return the identifier of the provisioned app service plan.

Let's now analyze the `app-service.json` template, as shown in Figure 4.26:

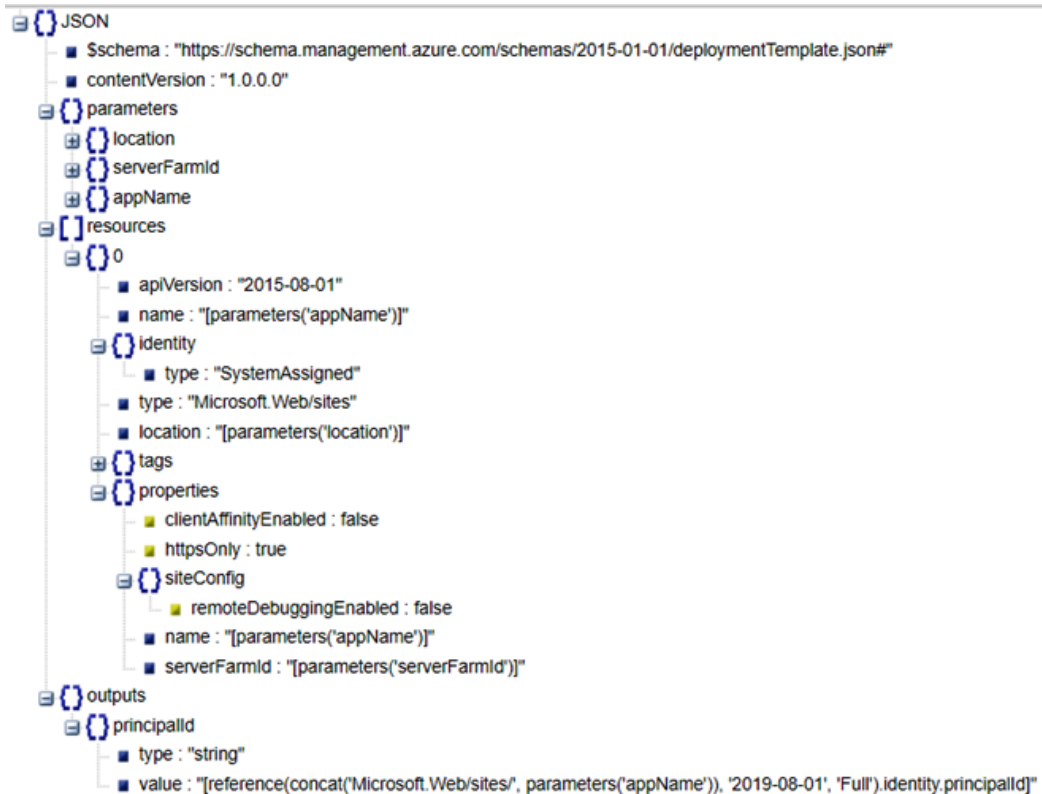


Figure 4.26 – An app service template

Again, *Figure 4.26* starts with the parameter section. This time, we expect to receive the `serverFarmId` parameter, which is the identifier of the app service plan that is attached to our app service. In the **Resources** section, we declare one resource of type **Microsoft.Web/sites**, and we enable a **managed system identity (MSI)** by defining a **SystemAssigned** identity. Note that this is a best practice that we will explore further in *Chapter 7, Security Architecture*. We also define a few configuration settings, and we attach our app service to the app service plan, through the `serverFarmId` parameter. Lastly, we output the identifier of the system identity, so that the pipeline can assign to it a number of permissions, if required. Admittedly, these two sample templates are oversimplified. They work, but in the real world, you would certainly deal with key vault secret references and more advanced app service settings.

Now that we have two separate templates, which can be reused across projects, we need our master file, which will link them for our demo project. Remember that only the master file is project-specific; linked templates are reusable.

Figure 4.27 shows you the contents of our master file:



Figure 4.27 – The master file

This time, let's jump directly to our **resources** section. Both resources are of the type **Microsoft.Resources/deployments**. The first one calls the `app-service-plan.json` linked template, which is in our storage account. It passes the name of the app service plan to the linked template. Our second resource calls our `app-service.json` linked template, and it passes the `serverFarmId` parameter, by using the output of our first deployment. Note that the **dependsOn** section tells the ARM to wait until the `app-service-plan` deployment is finished before starting our `app-service-deployment`.

To make a deployment dynamic, we work with parameter files or variables, which are passed during the deployment. That changes according to the target environment. For the sake of simplicity, we will work with the parameters shown in *Figure 4.28*:

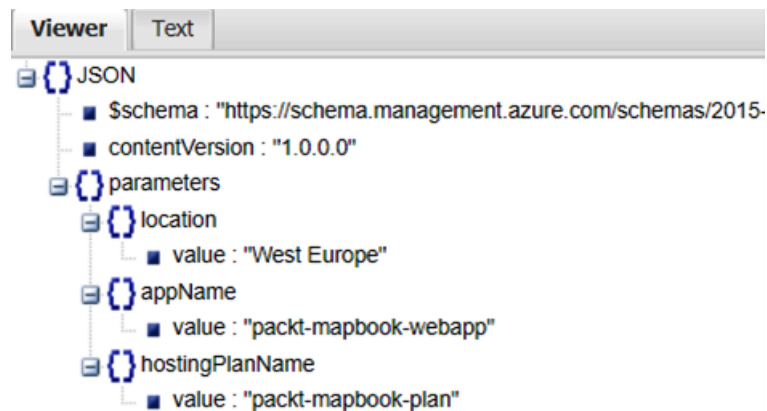


Figure 4.28 – The parameters

In a real-world deployment, both **appName** and **hostingPlanName** would vary, according to the target environment (prod/non-prod). Even the location parameter may vary when you're working with an active/passive **disaster recovery (DR)** setup.

We now have everything that we need to concretely provision these services into our *packt* resource group. We will use Azure DevOps (refer to the *Technical requirements* section to set up Azure DevOps if needed) to deploy our code. Once in Azure DevOps, you should do the following:

1. Create a repository named `packt` or reuse an existing one.
2. Make sure to replace the location of the storage account container in `master.json`. So, replace `https://packtmabook` with the storage account that you created previously.
3. Upload the `master.json` and `master.parameters.json` files into the `master` branch.

You should end up with the files shown in *Figure 4.29*:

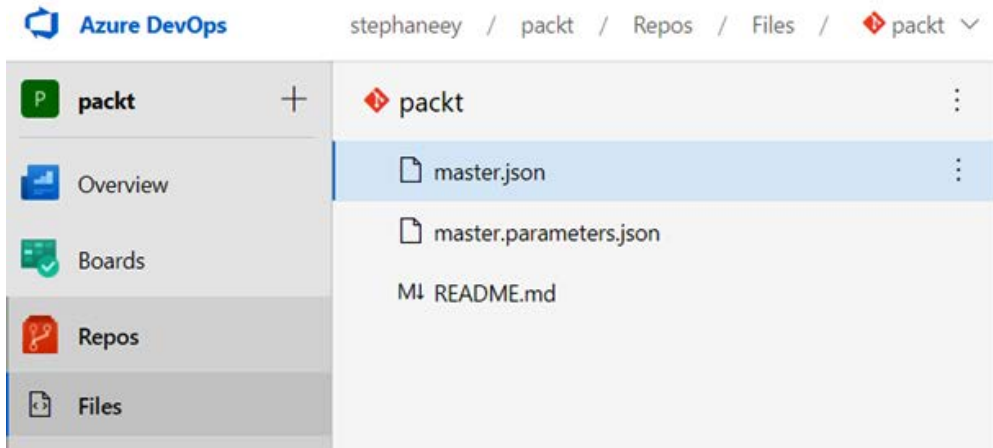


Figure 4.29 – The Azure DevOps repo

Remember that our shared templates are already in their storage account. In the last section of this chapter, we will show you how to link everything together directly in Azure DevOps.

Now we can create our YAML build pipeline to provision these resources. Locate the pipelines link in Azure DevOps and choose the Azure Repos Git template, as shown in *Figure 4.30*:

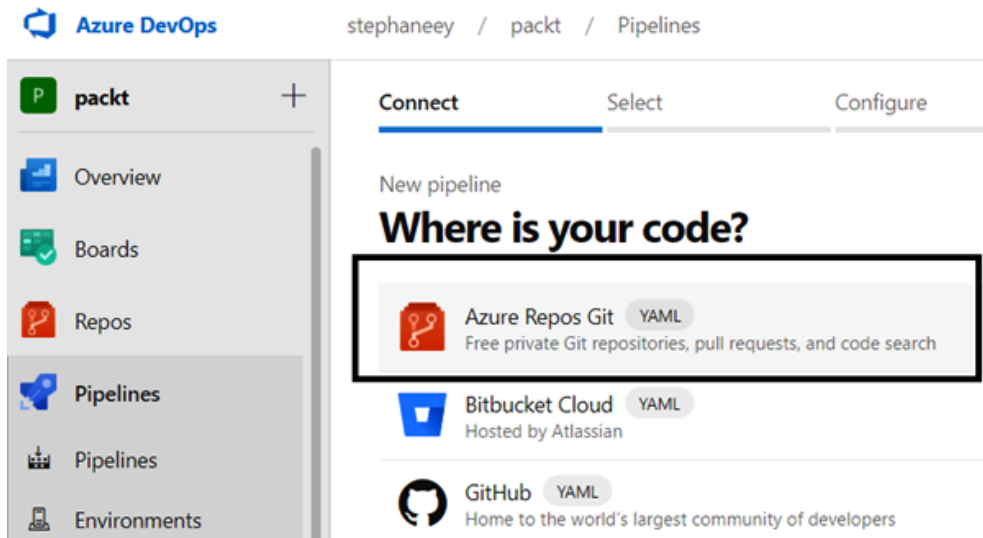


Figure 4.30 – Creating the build pipeline

- Next, we will select our **packt** repo, as shown in *Figure 4.31*:



Figure 4.31 – Our repository selection

- Then, we select the **Starter pipeline** option. See *Figure 4.32*:

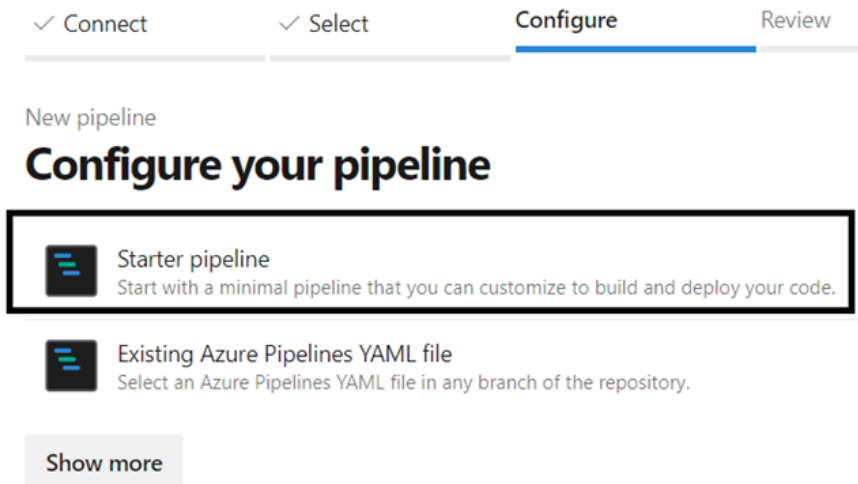


Figure 4.32 – Starter pipeline

- Click on **Show more** if you do not see the **Starter pipeline** option. You should end up with a `.yaml` file, as shown in *Figure 4.33*:

```

packt / azure-pipelines.yml *  [edit]

1 # Starter pipeline
2 # Start with a minimal pipeline that you can customize to build and dep
3 # Add steps that build, run tests, deploy, and more:
4 # https://aka.ms/yaml
5
6 trigger:
7   - master
8
9 pool:
10  - vmImage: 'ubuntu-latest'
11
12 steps:
13   - script: echo Hello, world!
14     displayName: 'Run a one-line script'
15
16   - script: |
17     echo Add other tasks to build, test, and deploy your project.
18     echo See https://aka.ms/yaml
19     displayName: 'Run a multi-line script'
20

```

Figure 4.33 – The starter pipeline YAML code

- Remove everything that is below `steps`. On the right-hand side, click on **Show assistant**. Then, search for the `arm` deployment task, as shown in *Figure 4.34*:

The screenshot shows the 'Review your pipeline YAML' interface in Azure Pipelines. The left pane displays the starter pipeline YAML code, with the `steps` section highlighted. The right pane shows a search for 'arm' tasks, listing two tasks: 'ARM template deployment' and 'AzSK ARM Template Checker'.

Figure 4.34 – Selecting the ARM deployment task

- Next, you must configure this task. See *Figure 4.35*:

← ARM template deployment ⓘ

Azure Details ^

Deployment scope * ⓘ

Resource Group ▾

Azure Resource Manager connection * ⓘ

Microsoft Azure Sponsorship(f63a908a-... ▾

Subscription * ⓘ

Microsoft Azure Sponsorship (f63a908a-... ▾

Action * ⓘ

Create or update resource group ▾

Resource group * ⓘ

packt ▾

Location * ⓘ

West Europe ▾

Template ^

Template location *

Linked artifact ▾

Template * ⓘ

master.json

Template parameters ⓘ

master.parameters.json

Override template parameters ⓘ

[About this task](#) [Add](#)

Figure 4.35 – Configuring the ARM deployment task

9. If you do not have an existing *Azure Resource Manager connection*, just make sure you're logged in with a user account that has sufficient permissions against the target Azure subscription to let Azure DevOps create a connection for you. This connection is used by the pipeline to authenticate against Azure Active Directory when deploying to the resource group. The YAML code of our configuration is shown in *Figure 4.36*:


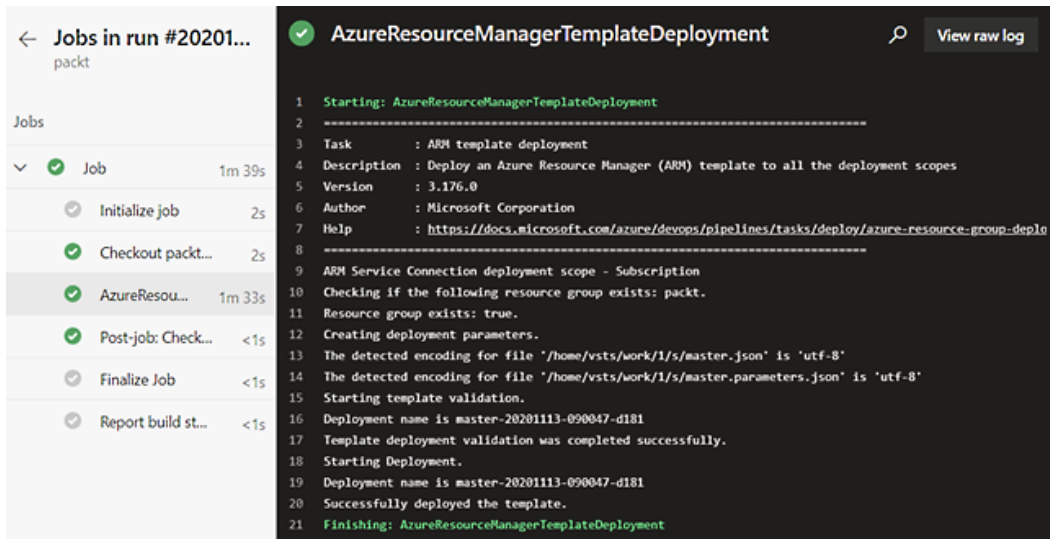
```
packt / azure-pipelines.yml *   
  
8  
9 pool:  
10   - vmImage: 'ubuntu-latest'  
11  
12 steps:  
    Settings  
13   - task: AzureResourceManagerTemplateDeployment@3  
14     inputs:  
15       deploymentScope: 'Resource Group'  
16       azureResourceManagerConnection: 'Microsoft Azure Sponsorship(f63a908  
17       subscriptionId: 'f63a908a-c054-4d45-a1fd-1eadaee67ffc'  
18       action: 'Create Or Update Resource Group'  
19       resourceGroupName: 'packt'  
20       location: 'West Europe'  
21       templateLocation: 'Linked artifact'  
22       csmFile: 'master.json'  
23       csmParametersFile: 'master.parameters.json'  
24       deploymentMode: 'Incremental'  
25
```

Figure 4.36 – The configuration YAML code

10. Now, if you click the **Save and run** button, you should be prompted to allow the pipeline to consume the service connection. *Figure 4.37* shows you what the job run looks like:



```

1 Starting: AzureResourceManagerTemplateDeployment
2 -----
3 Task      : ARM template deployment
4 Description : Deploy an Azure Resource Manager (ARM) template to all the deployment scopes
5 Version    : 3.176.0
6 Author     : Microsoft Corporation
7 Help       : https://docs.microsoft.com/azure/devops/pipelines/tasks/deploy/azure-resource-group-depl
8 -----
9 ARM Service Connection deployment scope - Subscription
10 Checking if the following resource group exists: packt.
11 Resource group exists: true.
12 Creating deployment parameters.
13 The detected encoding for file '/home/vsts/work/1/s/master.json' is 'utf-8'
14 The detected encoding for file '/home/vsts/work/1/s/master.parameters.json' is 'utf-8'
15 Starting template validation.
16 Deployment name is master-20201113-090047-d181
17 Template deployment validation was completed successfully.
18 Starting Deployment.
19 Deployment name is master-20201113-090047-d181
20 Successfully deployed the template.
21 Finishing: AzureResourceManagerTemplateDeployment

```

Figure 4.37 – The build job in action

11. Upon job completion (1 minute 39 seconds in our example), the resources should be provisioned in the target resource group. See *Figure 4.38*:

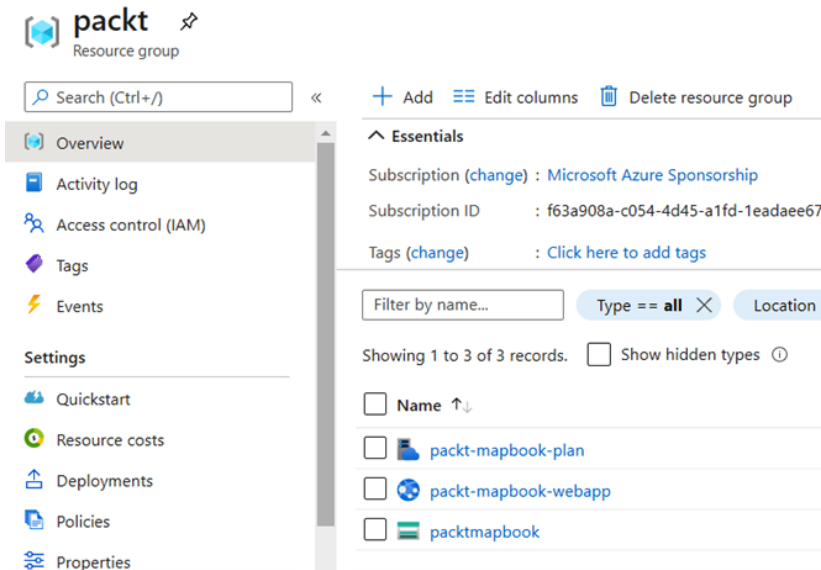


Figure 4.38 – The provisioned resources upon job completion

Note that in Azure DevOps, as in many factory tools, there are build and release pipelines. In our example, we have used a build pipeline to provision the services to the target environment. It has become a common trend to use YAML pipelines to do both build and release at the same time. Now that you have grasped the essential part of working with ARM templates, we will see how Azure Bicep may alleviate a few ARM constraints.

Getting started with Azure Bicep

At the time of writing, Azure Bicep is still in early development and not production ready yet, but this could change by the time you read this.

The main purpose of Azure Bicep is to do the following:

- Alleviate the complexity of the ARM template language to make it less verbose and to bring a more developer-friendly approach.
- Compiling Bicep files to produce a single ARM template. This prevents the use of a storage account or any other publicly available location for storing linked templates.

Unlike Terraform, Bicep remains Azure-specific. You can think of it as the next generation of the ARM language. To know more about Bicep and to stay tuned, you should subscribe to the Azure Bicep repo at <https://github.com/Azure/bicep>. Now we will redeploy exactly what we deployed previously, but we'll use raw ARM templates and evaluate the benefits of using Bicep. In order to perform this exercise, you must go through the following steps:

1. Install the Bicep client tools (get the setup instructions at <https://github.com/Azure/bicep/blob/main/docs/installing.md>).
2. Install the Visual Studio Code extension for Bicep. This step is optional, but it improves the authoring experience.

3. Once in Visual Studio Code, you can open the `packt.bicep` file, which is available in our GitHub repo. You should end up with something that looks like *Figure 4.39*:

```
packt.bicep > skuName
1 param hostingPlanName string
2 param appName string
3 param location string = resourceGroup().location
4 > param skuName string {
13 }
14 > param skuCapacity int {
18 }
19
20 resource plan 'Microsoft.Web/serverfarms@2020-06-01' = {
21   name: hostingPlanName
22   location: location
23   sku: {
24     name: skuName
25     capacity: skuCapacity
26   }
27 }
28
29 resource webapp 'Microsoft.Web/sites@2020-06-01' = {
30   name: appName
31   location: location
32   properties: {
33     serverFarmId: plan.id
34     siteConfig: {
35       remoteDebuggingEnabled: false
36     }
37   }
38 }
```

Figure 4.39 – Deploying an app service plan along with an app service using Bicep

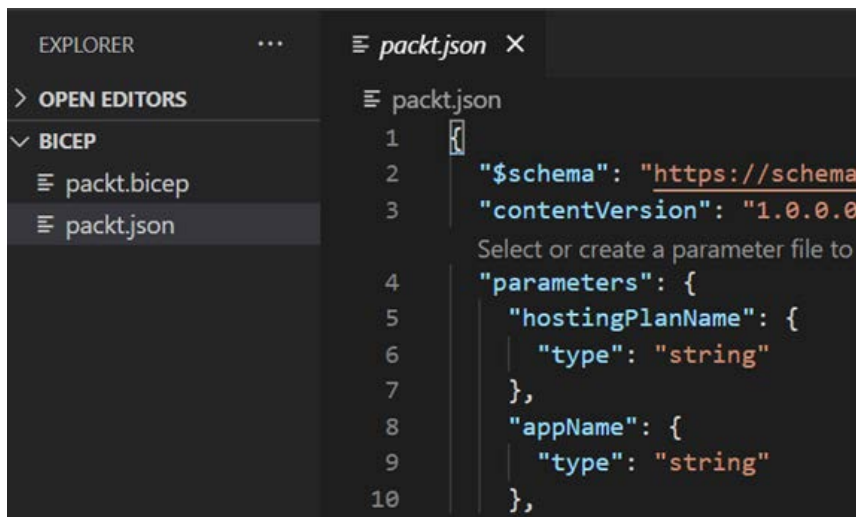
Notice how Bicep is much more concise than raw ARM templates. We first declare our parameters, and then our two resources. This time, the `webapp` resource does not need to explicitly declare its dependency on the `plan` resource. The simple reference `plan.id` is enough to let Bicep infer this dependency. This makes Bicep files much more readable.

However, at the time of writing, there is not yet an official Bicep integration with Azure DevOps (only some experimental features). So, we will simply build our Bicep file to get the resulting ARM template. *Figure 4.40* shows you how easy it is to build the Bicep file through the usage of the `bicep build` command:

```
35     remoteDebuggingEnabled:false
36   }
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\steph\packt\bicep> bicep build .\packt.bicep
PS C:\Users\steph\packt\bicep> |
```

Figure 4.40 – Building the Bicep file using Visual Studio Code's terminal

Upon build completion, you should get the corresponding ARM template generated within the same folder as your Bicep file, as shown by *Figure 4.41*:



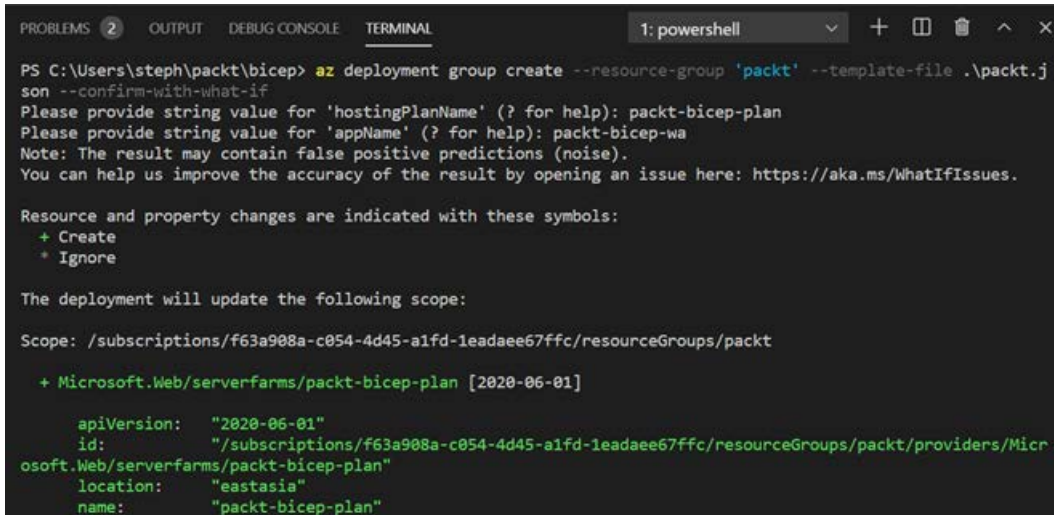
```
EXPLORER ... packt.json X
> OPEN EDITORS
  packt.json
  packt.bicep
  packt.json
BICEP
1
2
3
4
5
6
7
8
9
10
"$schema": "https://schema
$contentVersion": "1.0.0.0
Select or create a parameter file to
"parameters": {
  "hostingPlanName": {
    "type": "string"
  },
  "appName": {
    "type": "string"
  },
}
```

Figure 4.41 – The ARM JSON file generated by the Bicep build command

From there on, you can test the deployment with the `-what if` option of the Azure CLI, direct from Visual Studio Code. Opening the terminal again, you can enter the following command:

```
az deployment group create --resource-group 'packt' --template-
file .\packt.json --confirm-with-what-if
```


This command should validate the generated ARM template against the target resource group. You will be prompted to enter the hosting plan name, as well as the web app name, and you can get the result of that command, as shown by *Figure 4.42*:



```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
1: powershell
PS C:\Users\steph\packt\bicep> az deployment group create --resource-group 'packt' --template-file .\packt.j
son --confirm-with-what-if
Please provide string value for 'hostingPlanName' (? for help): packt-bicep-plan
Please provide string value for 'appName' (? for help): packt-bicep-wa
Note: The result may contain false positive predictions (noise).
You can help us improve the accuracy of the result by opening an issue here: https://aka.ms/WhatIfIssues.

Resource and property changes are indicated with these symbols:
+ Create
* Ignore

The deployment will update the following scope:

Scope: /subscriptions/f63a908a-c054-4d45-a1fd-1eadaee67ffc/resourceGroups/packt

+ Microsoft.Web/serverfarms/packt-bicep-plan [2020-06-01]

    apiVersion: "2020-06-01"
    id: "/subscriptions/f63a908a-c054-4d45-a1fd-1eadaee67ffc/resourceGroups/packt/providers/Micr
osoft.Web/serverfarms/packt-bicep-plan"
    location: "eastasia"
    name: "packt-bicep-plan"
  
```

Figure 4.42 – Validating the generated ARM template against Azure

Note that you need to have the latest version of the Azure CLI installed on your machine in order to test the deployment with the `-confirm-whatif` option. If you want to test this, an alternative is to upload the generated ARM template into an Azure DevOps repo, and then deploy it, like we did in the previous section. Now we are going to see how Terraform compares with ARM templates and Bicep.

Getting started with Terraform

Terraform is HashiCorp's star IaC product. They define it as a tool that can be used to provision and manage any cloud, infrastructure, or service. Terraform's architecture is based on hundreds of providers, among which is the Azure provider. The value proposal of Terraform is to propose a common way to define IaC templates, no matter the target platform. The least we can say is that HashiCorp succeeded in making a very broad and good product. In addition to the official providers, there are also dozens of community-provided providers. Terraform's most important commands are the following:

- `init`: Used only when referencing a new provider or a new provider version
- `plan`: An optional step for comparing the known state with the new desired state
- `apply`: A command to apply the new desired state for the resources defined in the template

Templates are written in **HCL (HashiCorp configuration language** – <https://www.terraform.io/docs/configuration/syntax.html>). We are going to replay the resource deployment we made earlier (with ARM templates and Bicep) to see how it compares. Terraform is easy to set up because there is only a single binary file (which you can find on Terraform's download page). However, we will make it even easier for you to experiment by using the Azure Cloud Shell that we have already used with the imperative command-line tools. Start by launching Azure Cloud Shell, as we did earlier. Once you're in Cloud Shell, run the following command:

```
mkdir terraform
```

Next, run this command:

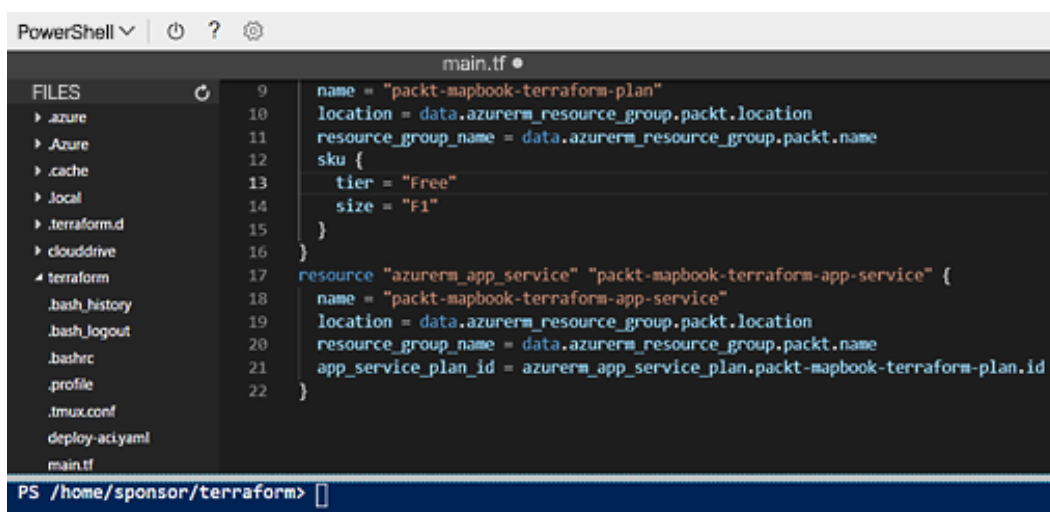
```
cd terraform
```

Then, you can launch the default Cloud Shell editor using the following command:

```
code main.tf
```

This allows us to create our `main.tf` template in the current folder. Once there, you can copy the contents of our sample Terraform file, which is located at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/tree/master/Chapter04/IaC/terraform/main.tf>.

You should end up with the Terraform template looking like it does in *Figure 4.43*:

The image shows a screenshot of the Azure Cloud Shell interface. At the top, it says "PowerShell" with a dropdown arrow, a power icon, a question mark, and a gear icon. Below this is a file explorer on the left with a refresh icon, listing files: .azure, .Azure, .cache, .local, .terraform.d, clouddrive, terraform (selected), .bash_history, .bash_logout, .bashrc, .profile, .tmux.conf, deploy-aci.yaml, and main.tf. The main area shows the content of main.tf with line numbers 9 through 22. The code defines an Azure App Service plan and an App Service resource. The App Service plan is named "packt-mapbook-terraform-plan" and is configured with tier "Free" and size "F1". The App Service resource is named "packt-mapbook-terraform-app-service" and is linked to the App Service plan.

```
9  name = "packt-mapbook-terraform-plan"
10 location = data.azurem_resource_group.packt.location
11 resource_group_name = data.azurem_resource_group.packt.name
12 sku {
13   tier = "Free"
14   size = "F1"
15 }
16 }
17 resource "azureram_app_service" "packt-mapbook-terraform-app-service" {
18   name = "packt-mapbook-terraform-app-service"
19   location = data.azurem_resource_group.packt.location
20   resource_group_name = data.azurem_resource_group.packt.name
21   app_service_plan_id = azureram_app_service_plan.packt-mapbook-terraform-plan.id
22 }
```

Figure 4.43 – The Terraform template in Azure Cloud Shell

Let's now analyze this template. The first block, as illustrated in *Figure 4.44*, is the declaration of the Azure provider:

```
provider "azurerm" {
  version = "2.36.0"
  features {}
}
```

Figure 4.44 – The Azure provider declaration

This is where you will have to use the `init` command later for Terraform to pull the required resources for the `azurerm` module. Our second code block, shown in *Figure 4.45*, is the declaration of our existing *packt resource group*:

```
data "azurerm_resource_group" "packt" {
  name = "packt"
}
```

Figure 4.45 – The declaration of the packt resource group

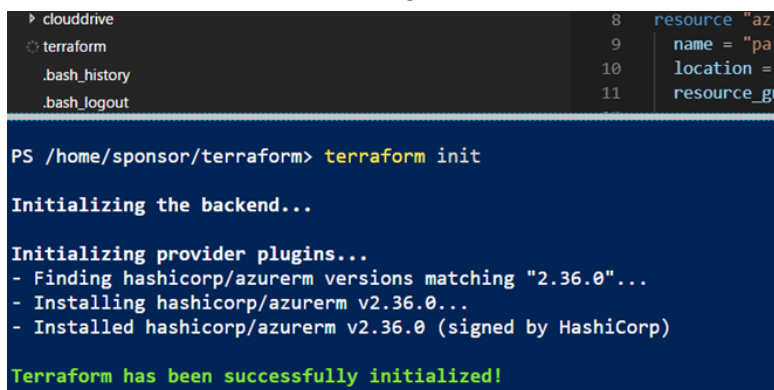
We use the **data** construct to refer to any resource that is not present in the template itself. Our next two blocks, shown in *Figure 4.46*, are the two resources we want to provision:

```
resource "azurerm_app_service_plan" "packt-mapbook-terraform-plan" {
  name = "packt-mapbook-terraform-plan"
  location = azurerm_resource_group.packet.location
  resource_group_name = azurerm_resource_group.packet.name
  sku {
    tier = "Free"
    size = "F1"
  }
}

resource "azurerm_app_service" "packt-mapbook-terraform-app-service" {
  name = "packt-mapbook-terraform-app-service"
  location = azurerm_resource_group.appservice-rg.location
  resource_group_name = azurerm_resource_group.appservice-rg.name
  app_service_plan_id = azurerm_app_service_plan.packt-mapbook-terraform-plan.id
  tags = {
    environment = var.environment
  }
}
```

Figure 4.46 – The resources to provision

The first resource is our app service plan, and the second resource is our app service. Notice the reference for the `app_service_plan_id` property. We can easily reference our first resource from the second one. For the sake of brevity, we do not show you how to work with the parameter files, nor the outputs, but Terraform also supports them. Now we are ready to try this template out. Back in Azure Cloud Shell, you must first run the `terraform init` command, as shown in *Figure 4.47*:



```

8 resource "az
9 name = "pa
10 location =
11 resource_gr

PS /home/sponsor/terraform> terraform init

Initializing the backend...

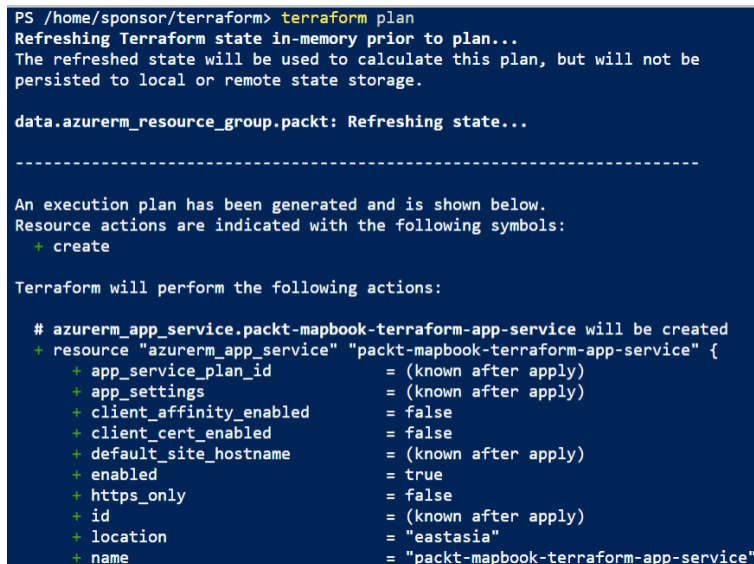
Initializing provider plugins...
- Finding hashicorp/azurerm versions matching "2.36.0"...
- Installing hashicorp/azurerm v2.36.0...
- Installed hashicorp/azurerm v2.36.0 (signed by HashiCorp)

Terraform has been successfully initialized!

```

Figure 4.47 – The terraform init command

This allows Terraform to resolve all dependencies to the Azure provider. Next, we can optionally run the `terraform plan` command to evaluate the changes to the existing resources. See *Figure 4.48*. Remember that this is our first deployment, so Terraform has no recorded state yet:



```

PS /home/sponsor/terraform> terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

data.azurem_resource_group.packt: Refreshing state...

-----

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# azurerm_app_service.packt-mapbook-terraform-app-service will be created
+ resource "azurerm_app_service" "packt-mapbook-terraform-app-service" {
+ app_service_plan_id      = (known after apply)
+ app_settings             = (known after apply)
+ client_affinity_enabled = false
+ client_cert_enabled      = false
+ default_site_hostname    = (known after apply)
+ enabled                  = true
+ https_only               = false
+ id                       = (known after apply)
+ location                 = "eastasia"
+ name                     = "packt-mapbook-terraform-app-service"

```

Figure 4.48 – The terraform plan command

You can review the complete output, but since the template is very basic, everything should be fine. Finally, we concretely deploy the template with `terraform apply`, as shown in *Figure 4.49*:

```
PS /home/sponsor/terraform> terraform apply -auto-approve
data.azure_rm_resource_group.packt: Refreshing state...
azure_rm_app_service_plan.packt-mapbook-terraform-plan: Creating...
azure_rm_app_service_plan.packt-mapbook-terraform-plan: Still creating... [10s elapsed]
azure_rm_app_service_plan.packt-mapbook-terraform-plan: Creation complete after 13s [id=/subscriptions/packt/providers/Microsoft.Web/serverfarms/packt-mapbook-terraform-plan]
azure_rm_app_service.packt-mapbook-terraform-app-service: Creating...
azure_rm_app_service.packt-mapbook-terraform-app-service: Still creating... [10s elapsed]
azure_rm_app_service.packt-mapbook-terraform-app-service: Still creating... [20s elapsed]
azure_rm_app_service.packt-mapbook-terraform-app-service: Still creating... [30s elapsed]
azure_rm_app_service.packt-mapbook-terraform-app-service: Still creating... [40s elapsed]
azure_rm_app_service.packt-mapbook-terraform-app-service: Still creating... [50s elapsed]
azure_rm_app_service.packt-mapbook-terraform-app-service: Creation complete after 58s [id=/subscriptions/packt/providers/Microsoft.Web/sites/packt-mapbook-terraform-app-service]

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
PS /home/sponsor/terraform>
```

Figure 4.49 – Deploying resources with `terraform apply`

Figure 4.49 shows that the `apply` command gives you a nice output, given what has been done and how long each step took to complete. The experience is quite similar to Azure Bicep, which was probably inspired by Terraform.

Once you have applied the template, the resources should be available in the resource group. Now that we have provisioned our two resources, Terraform has generated a state file, as shown in *Figure 4.50*:

```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
PS /home/sponsor/terraform> ls
main.tf terraform.tfstate
PS /home/sponsor/terraform> █
```

Figure 4.50 – The state file

Terraform maintains the deployment state in this state file. If you are working for a small company that does not want to invest money in a factory, and where you are *the* IaC person, then you can rely on Azure Cloud Shell alone and have your own local state files. If you work as a team with a real factory, such state files must be persisted in **remote backend stores**, such as Azure Storage.

You can inspect the state file, but you should not change it manually. This is the source of truth for Terraform. Any change to the deployed resource, via another channel, can be detected by Terraform when running the plan command. To test this, you can switch the app service's **HTTPS only** property toggle to **on**, as shown in *Figure 4.51*:

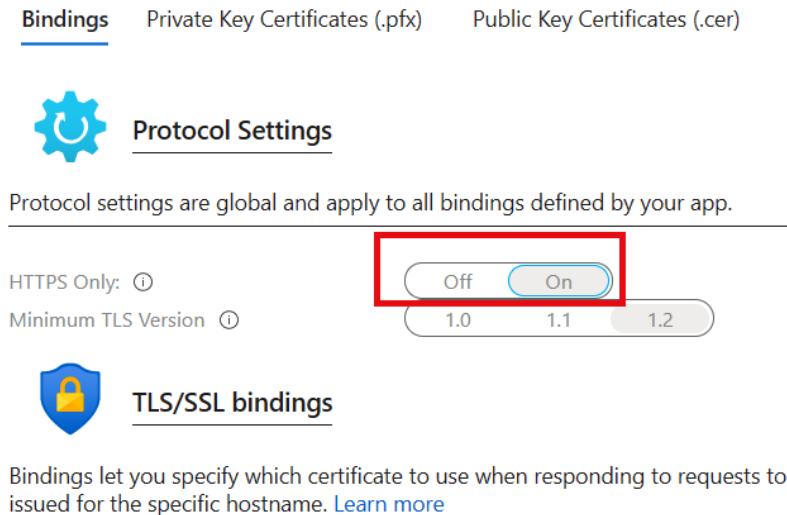


Figure 4.51 – Switching HTTPS only from off to on

Now that we have made this change through the Azure Portal, instead of Terraform, if we rerun a `plan` command, we see that the current state has deviated from Terraform's state. See *Figure 4.52*:

```

client_cert_enabled = false
default_site_hostname = "packt-mapbook-
enabled = true
~ https_only = true -> false
id = "/subscriptions
ook-terraform-app-service"
location = "eastasia"
name = "packt-mapbook-
outbound_ip_addresses = "13.75.34.175,1
possible_outbound_ip_addresses = "13.75.34.175,1

```

Figure 4.52 – A change detected by the Terraform plan command

Terraform has flagged the `https_only` property, and it shows that the current value is `true`. This will be set back to `false` if we run the `apply` command. In case you cannot avoid changes from channels other than Terraform, it is possible to refresh the Terraform state. Something important to note as well is the fact that, unlike Bicep, Terraform is not limited to the possibilities of ARM templates because Terraform also makes use of the Azure CLI behind the scenes. For instance, while ARM templates cannot deal with Azure Active Directory app registration (a very common need), Terraform can. Also, it is important to note that Terraform natively supports the deployment of ARM templates by using *HCL* in case a feature is available in ARM but not in Terraform.

To conclude, we can say that Terraform is indeed easy. It runs across the different providers and is more powerful than ARM templates and Azure Bicep. You should definitely favor Terraform over ARM and Azure Bicep if you know that you will not be restricting your IaC practice to Azure alone. Otherwise, it is more a question of style and preference as opposed to a genuine black or white answer.

Let's now explore a real-world approach – working with native ARM templates in an industrial manner.

Zooming in on a reference architecture with Azure DevOps

So far, we have reviewed the fundamentals of Terraform, ARM templates, and Azure Bicep. It is now time to see how you can concretely set up a factory that's designed to provision resources and deploy applications in an industrial manner. Of course, we will not walk you through the complete setup, but we will describe the possible approaches.

Beware that it takes time to get a fully industrialized factory up and running, and it is a significant investment. So far, we have largely focused on the IaC bits, but of course the infrastructure components that we provision are used by applications that have their own life cycle. At the end of the day, you need to find a way to deploy both the application code and the infrastructure together, while still being able to test your own infrastructure work, independently of the applications that will consume your components. Therefore, we must distinguish the authoring and versioning of the IaC components themselves (our shared templates), and the project-specific pipelines that will consume these IaC artifacts. Our objectives are as follows:

- We want to be autonomous to define our standard templates.
- We want to be autonomous to test our standard templates.
- We want applications to consume officially released templates.

- We do not want to cause any regression to existing applications, so we need to version our ARM templates to support multiple versions at a time, and to be backward compatible.

We will now describe two approaches – a simple one and a more advanced one. Why two? Simply because not every company has a significant number of assets in Azure to justify a very advanced factory, which requires dedicated manpower to work on it. Let's start with the simple approach.

Using a simple approach to an IaC factory

Since ARM templates are fully declarative, they do not require an effective build prior to consumption. You may simply store your ARM template files into different Git branches to version them, as shown in *Figure 4.53*:

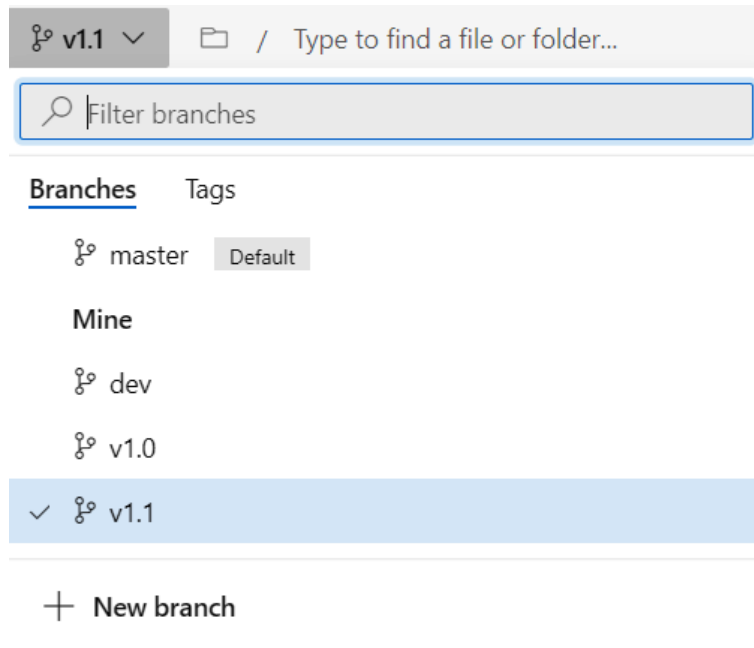


Figure 4.53 – Using branches to version ARM templates

Once your changes have been tested and validated in the DEV branch, you merge it with the master branch and create a new release branch from the master to publish a new version. For non-breaking changes, you may override an existing version and merge that one with the master branch. Then you simply publish the whole branch onto a shared storage account in Azure, which can be used by the various applications. *Figure 4.54* shows the different steps involved in that process:

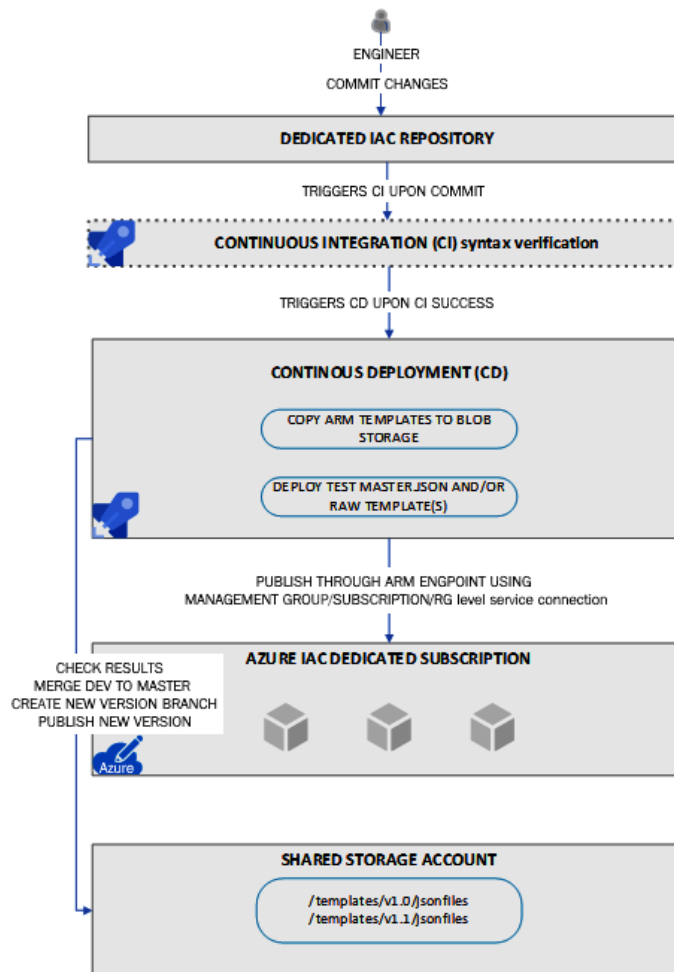


Figure 4.54 – A simple approach for IaC components

Firstly, we work with a dedicated, separate Azure DevOps repository. Whenever an engineer commits a change to an ARM template, it triggers an optional CI build that checks the syntax before trying a deployment to Azure. During deployment, you will copy the ARM templates to a test storage account (just for you). You will deploy the ARM templates independently and/or with a master file, which assembles them to mimic a real application. Once the validation has been done, you can simply publish the new version to a new branch, and then publish the branch to a shared storage account, which is used by the applications. This approach is very easy, but it has the following drawbacks:

- A new version branch will be created whenever any template gets modified. So, if you have 10 services and only change one of them, you will create a new branch that will contain 9 unchanged services, plus the changed one. While this is a bit of a waste of space/resources, ARM templates are very lightweight, so the impact is not dramatic.
- Similarly, whenever a single template changes, all the templates will be published to the shared storage account, in a dedicated container. It's the same consequence and observation as before. You are charged for the storage costs, but the templates are so tiny that you should not even really notice the difference.

Now that you have published your templates, they are ready for consumption. *Figure 4.55* shows you how to consume them easily:

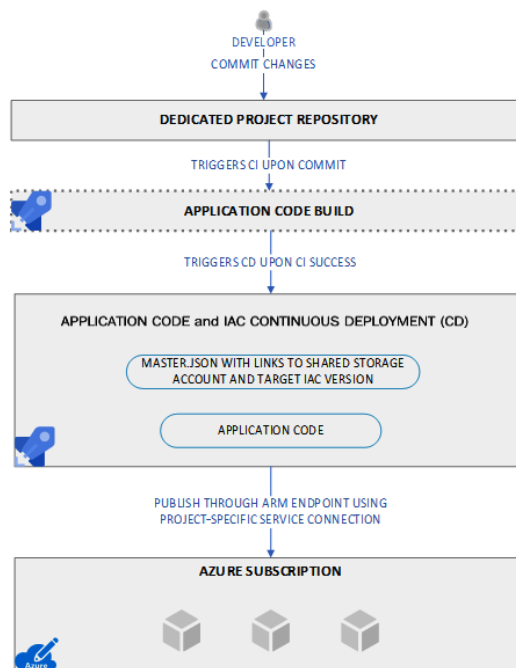


Figure 4.55 – IaC consumption – Simple approach

This time, whenever the application code changes, there is a CI build that builds the application code, followed by a CD that deploys both the infrastructure and the application code together. Notice that the application pipeline references the `master.json` file, where it assembles the components that are needed for the application.

In the master file, you must reference the shared storage account and the target IaC version. Should new IaC versions be released, you will not be impacted. Should non-breaking changes overwrite the IaC version you are using, you will adjust the application-specific infra components during the next application deployment. Remember that ARM templates are idempotent, so you can redeploy against an existing application at no risk. The only drawback of redeploying the infrastructure is the loss of time, which is taken by the ARM verification, but this is usually OK.

The simple approach is rather easy to implement and may work with non-dedicated DevOps specialists. Let's now look at a more advanced approach.

Using an advanced approach to an IaC factory

If you have more people available to work on the factory, and if you are using a large number of Azure services, you may want to go the extra mile and use IaC artifacts as shared code libraries, as you would typically do with pure application code. You may also want to spare the extra build minutes and avoid redeploying the ARM templates at every deployment. A few lost minutes here and there are OK, as long as you do not make 300 deployments a day! You might also want to have more granularity in the IaC components that you want to use, such as, for example, mixing different service versions (version 1.1 of `storage-json` with version 1.2 of `apim-json`), which is not possible with the simple approach we depicted earlier. *Figure 4.56* shows you the steps involved in the advanced approach:

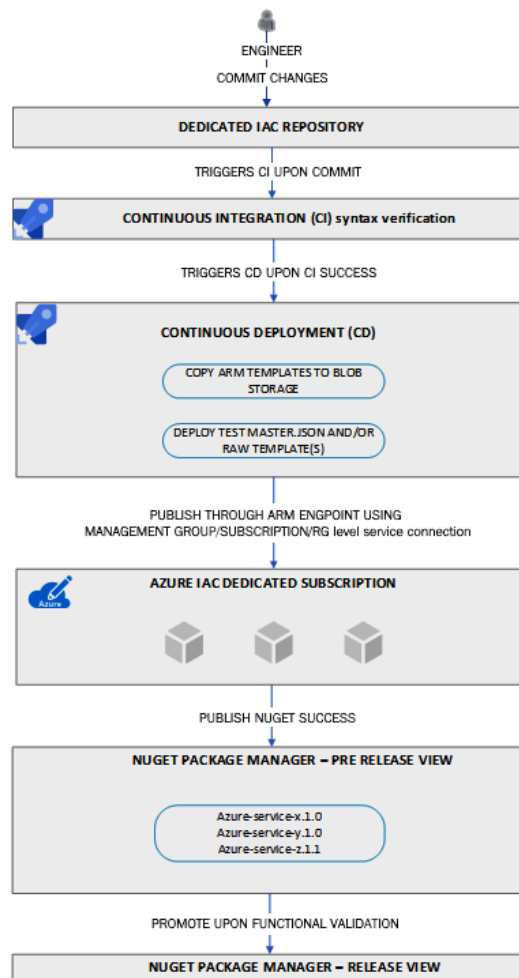


Figure 4.56 – An advanced approach for IaC component industrialization

As with the simple approach, a CI build kicks off upon a commit over a template, which in turn triggers the CD step. Resources are also being deployed to Azure. Upon successful deployment, the template is published to Azure DevOps Artifacts as a pre-release view, as a NuGet package. Following a functional validation of the pre-release, you can promote the pre-release view to a release view. Azure DevOps exposes a REST API to do the promotion, but you can also find an extension (<https://marketplace.visualstudio.com/items?itemName=rvo.vsts-promotepackage-task>) on the Azure DevOps marketplace for doing so. Once promoted, the packages are available for consumption.

Figure 4.57 shows you the consumption steps:

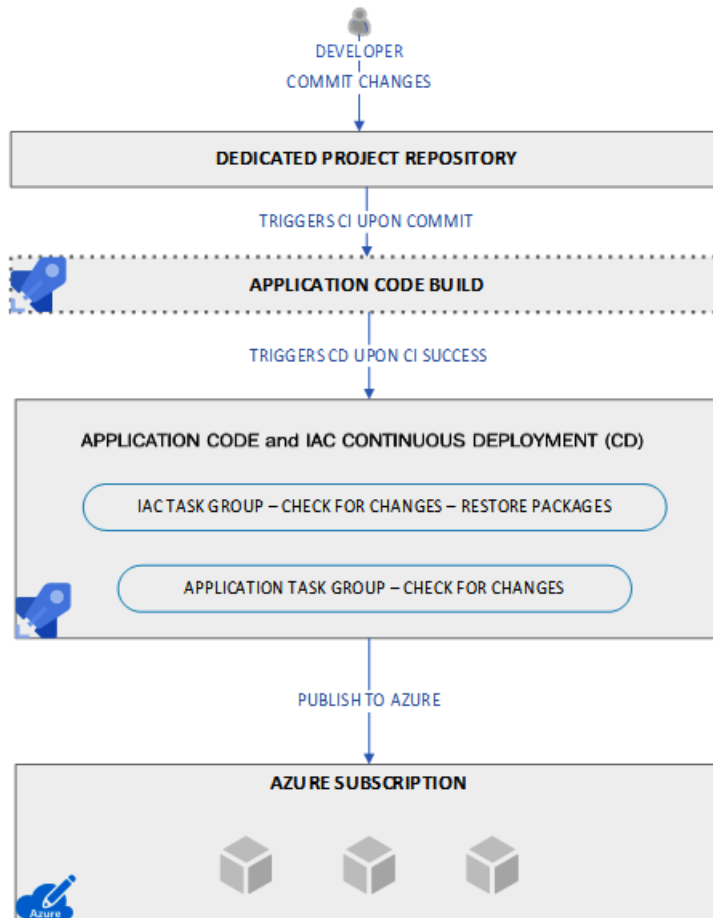


Figure 4.57 – IaC advanced – Consumption

The steps are very similar to the simple approach, except for two noticeable differences:

- We validate that re-provisioning of the resources is required, against the already existing deployed resources. You can do that using Azure resource tags to compare the versions. In case we need to redeploy any IaC component, we'll restore their corresponding NuGet packages from Azure Artifacts and we'll copy their contents to a blob storage, to let the ARM endpoint pull the templates from that location. We will (re)provision and tag the Azure resources accordingly.
- Similarly, applications often deal with multiple build artifacts. Here again, we will only redeploy if needed.

This slightly different approach might seem trivial, but you cannot achieve this without writing custom code/tasks, while our simple approach can be fully achieved with out-of-the-box Azure DevOps tasks. Let's now recap what we have seen during this chapter.

Summary

In this chapter, we briefly explained what the benefits are of a CI/CD pipeline and of DevOps tooling in general. We also highlighted how challenging it can be to have a fully functioning factory.

We made it clear that there is only one ruler in Azure's IaC world: the ARM endpoint. All the imperative and declarative tools and languages ultimately talk to the ARM endpoint. We shed some light on Terraform, native ARM templates, and Azure Bicep, ARM's next-generation language. Beyond imperative tools and declarative languages, we explored how to set the different elements to music with Azure DevOps. Concepts explained for Azure DevOps also apply to other platforms. By now, you should have understood that a fully functioning factory leveraging IaC is a vital element for building and provisioning cloud solutions. You should have gained sufficient knowledge to get started with your own factory, or to accompany DevOps teams on their CI/CD journey.

IaC is a key part of pure cloud and cloud-native applications, which we are going to explore in the next chapter.

Section 2: Application Development, Data, and Security

Building cloud and cloud-native solutions means that you must be able to deal with heavily distributed architectures. We will guide you through some typical distributed patterns and how the ecosystem (Azure, K8s) can help you achieve more and better results. Building solutions does not solely mean building applications. It may also refer to pure data solutions, which you need to extract valuable insights and to help decision-makers make informed decisions. We will review the numerous data services and shed some light on how to combine traditional and modern data practices in Azure. Finally, whatever you build, it should be secure, so we will also focus on this very important dimension: security in the cloud.

We will cover the following topics in this section:

- *Chapter 5, Application Architecture*
- *Chapter 6, Data Architecture*
- *Chapter 7, Security Architecture*

5

Application Architecture

In the previous chapters, we thoroughly covered infrastructure. Now, the time has come to discuss your application-related concerns. After all, the purpose of an infrastructure is to host one or more applications. We will only consider **Communication as a Service (CaaS)**, **Platform as a Service (PaaS)**, and **Function as a Service (FaaS)**, which are cloud and cloud-native approaches, in line with modern development techniques. We will not delve into **Infrastructure as a Service (IaaS)**, since it has zero impact on the way applications should be architected, compared to on-premises data centers.

In this chapter, you will learn about the difference between traditional IT practices and cloud-native practices. Additionally, you will learn about the distributed nature of both cloud and cloud-native solutions. Modern solutions are less code-centric because many typical application duties can be offloaded to off-the-shelf services. With serverless, you can even build powerful zero-code solutions.

More specifically, we will cover the following topics in this chapter:

- Understanding cloud and cloud-native development
- Exploring the Azure Application Architecture Map and cloud design patterns
- Exploring **Event-Driven Architectures (EDAs)** and messaging architectures
- Developing microservices

The prevalence of cloud ecosystems is a game changer, and it should be tackled from the beginning when you're designing applications. The application architect shouldn't only focus on the application code and some development design patterns. As per our real-world observations, this fact is often misunderstood or overlooked by application architects, who tend to forget about the ecosystem their code is running in. By the end of this chapter, you will be better equipped to grasp new distributed developments and modern ecosystems.

Let's walk through this world that's in a constant state of evolution!

Technical requirements

If you want to practice the explanations provided in this chapter, you will require the following:

- **An Azure subscription:** To create your free Azure account, follow the steps explained at <https://azure.microsoft.com/free/>.
- **Visual Studio 2019:** You will need this to open the solution provided on GitHub.
- **Dapr (Distributed Application Runtime) CLI:** To install the Dapr CLI, follow the steps explained at <https://github.com/dapr/cli>.
- **Docker:** To install Docker Engine, follow the steps explained at <https://docs.docker.com/get-docker/>.
- **Microsoft Visio:** You will need this to open the diagrams, although they are also provided as PNG files.
- **Fiddler or Postman:** Use your preferred tool. We use Fiddler in this book, but feel free to use any other HTTP tool you want.

All of the code samples and diagrams are available at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/tree/master/Chapter05/>.

The CiA videos for this book can be viewed at: <http://bit.ly/3pp9vIH>

Understanding cloud and cloud-native development

Before going further into the substance of the map, let's introduce the notion of cloud and cloud-native development, which you might not be familiar with. The cloud is a particular ecosystem, and we should ideally leverage its unique capabilities to minimize frictions and frustrations. The best way to achieve this goal is to develop cloud and cloud-native applications. Developing cloud and cloud-native solutions also transforms the way we design applications and the type of frameworks we might use or build, and that is what we are going to explore in this section.

One of the biggest challenges when starting to work with cloud and cloud-native approaches is the fact that applications are distributed. In traditional IT, we still have many monoliths, or in the best case, we work with **Service-Oriented Architecture (SOA)**. SOA has proven its value and is certainly still future proof; however, as always, the IT world has evolved, and new paradigms have emerged.

Our current application perimeter is often restricted to an N-tier architecture, with a frontend, a backend, and a database, living on a few servers. We control the entire code base and underlying systems because we do everything ourselves. We write custom in-house frameworks to deal with cross-cutting concerns, such as logging, exception handling, and more. Our in-house frameworks become monoliths themselves. If you want, you can still work like that in the cloud, using the IaaS hosting model. As mentioned in *Chapter 1, Getting Started as an Azure Architect*, IaaS is a perfect *business-as-usual* solution. You keep controlling everything, except the hardware itself, and you can keep developing assets in a traditional way. We also explained that you might still have some fruitful business cases, but this is certainly not cloud nor cloud native. Of course, IaaS can be used as a transition to first lift-and-shift your assets as is until they reach their end of life, or until you have a budget to refactor and transform them into modern applications.

With cloud and cloud-native applications, we tend to rely more on off-the-shelf frameworks, services, and ecosystems. We assemble existing cloud services together and add our code on top. Some services come with a lot of built-in functionality. For example, the **Azure Cognitive Services** offerings are feature-rich, which prevents you from writing everything yourself, and it helps you to build applications faster. Another example is **Azure API Management**, which ships with many technical features, such as API throttling and **JSON web token (JWT)** token validation, which we can use to our advantage, without reinventing the wheel in the code. Moreover, such services are designed with resilience and robustness from the ground up. You will, at best, create a pale imitation, that is, if you stubbornly develop it yourself.

To adopt a cloud or cloud-native approach, you inevitably need to modernize/refactor the assets or start from a greenfield. Although there is probably not a single definition of cloud and cloud-native development, let's share our vision and try to demystify it. A **cloud-development approach** looks similar to *Figure 5.1*:

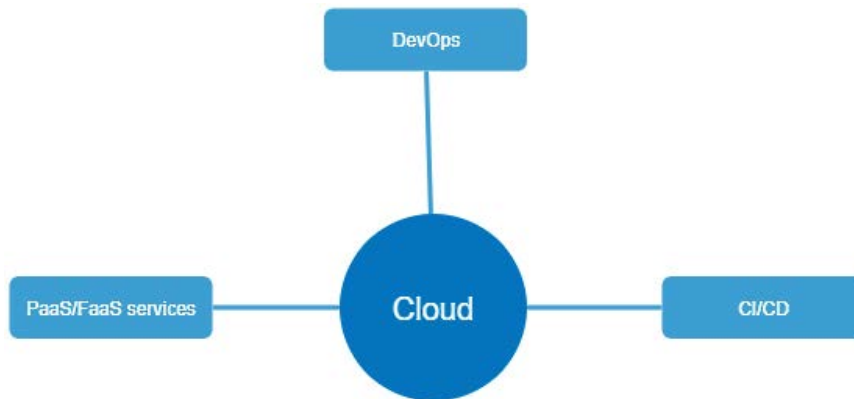


Figure 5.1 – Cloud development

Cloud development relies on three pillars:

- **DevOps:** This refers to the way we organize teams to collaborate on a product. Beyond technicalities, it really focusses on the organizational aspects, for instance, the notion of virtual teams, the product owner, the **minimum viable product (MVP)**, user stories, sprints, and more.
- **CI/CD:** We explained CI/CD in *Chapter 4, Infrastructure Deployment*. As a reminder, it is our factory, that is, our automation tool chain that plays a crucial role to set things to music. This infrastructure-as-code approach is an integral part of cloud and cloud-native development.
- **PaaS/FaaS:** We explained these hosting models in *Chapter 1, Getting Started as an Azure Architect*. The point we want to make here is that you can delegate a big part of the functionalities and technicalities to off-the-shelf services that are managed by the cloud provider. You accept that you do not control everything, and you avoid reinventing the wheel in your code. As an application architect, you must study the Azure service catalog and see what it can directly bring to your application. You may want to add some abstraction layers in order to prevent vendor lock-in syndrome, but you should not abuse them.

A **cloud-native development approach** looks similar to *Figure 5.2*:

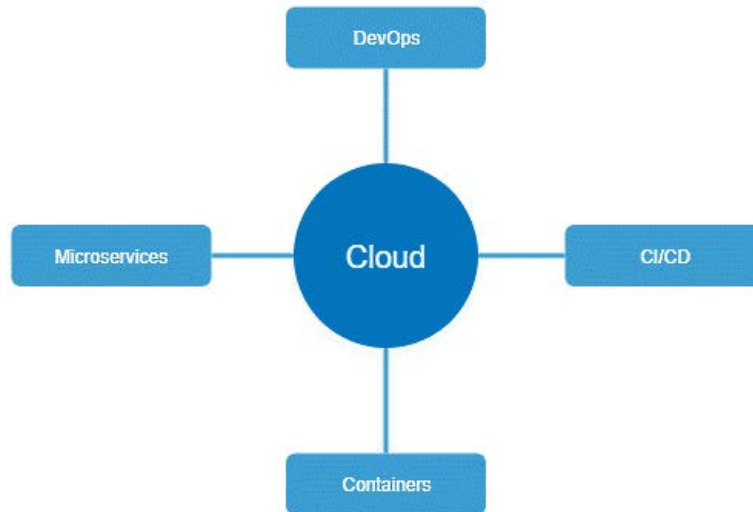


Figure 5.2 – Cloud-native development

DevOps and CI/CD remain part of the equation, but we add microservices and containers to the mix. We demonstrated in *Chapter 3, Infrastructure Design*, why using **Kubernetes (K8s)** is a better fit for microservices than native Azure services. Paradoxically, it appears that you can make cloud-native development on-premises, because it is, of course, possible to self-host container orchestrators. In theory, shipping code in containers should not make any difference, other than packaging the application differently. Reality proves otherwise. Make no mistake, this is a world of difference.

Every serious container platform relies on an orchestrator (K8s, OpenShift, and more), and such platforms come with their own ecosystems. They include plenty of tools and services that you cannot ignore as an application architect because they precisely encompass both technical and functional features. Let's now dive into our map, focusing on the cloud design patterns. We will also describe some of the PaaS/FaaS services that are part of a cloud development approach.

Exploring the Azure Application Architecture Map

In this section, we are going to explore the Azure Application Architecture Map. The purpose of this map is to help you find relevant services, with regard to cloud and cloud-native design patterns. It also browses the different data options for BASE and ACID database engines because data is part of every application.

In *Chapter 1, Getting Started as an Azure Architect*, we saw that the application architect mainly focuses on the programming languages, **Software Development Kit (SDKs)**, and design patterns in general. As a metaphor, we could say that application architects remain at layer 7 of the **Open Systems Interconnection (OSI)** model. Therefore, we will review these patterns, as well as some of the useful libraries you can use in your applications. To remain on the Microsoft ecosystem, we will mostly list .NET Core libraries, although most of them are also available in other programming languages:

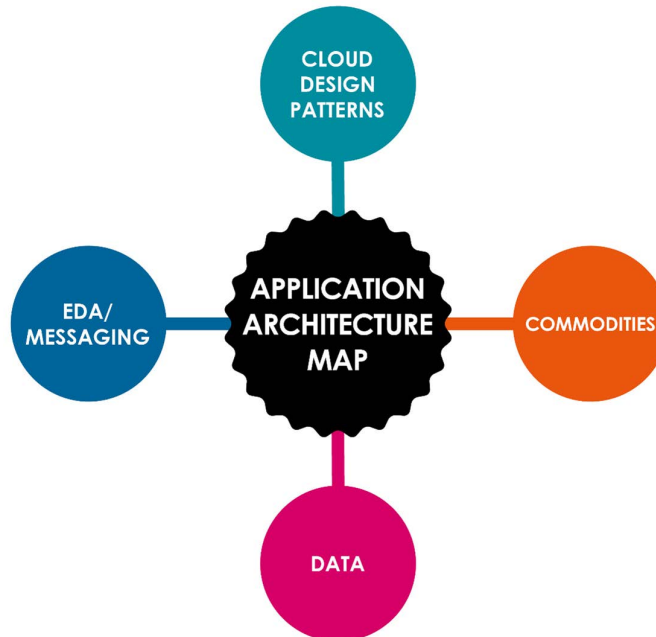


Figure 5.3 – The Azure Application Architecture Map

Important note

To view the full Azure Application Architecture Map (*Figure 5.3*), you can download the PDF file here: <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/blob/master/Chapter05/maps/Azure%20Application%20Architecture.pdf>.

Our map has four top-level groups:

- **DATA:** Every application deals with data. We will mostly cover Cosmos DB and its prevalence over all the other database engines, when dealing with **CQRS** and **Event Sourcing**.

- **CLOUD DESIGN PATTERNS:** This refers to an enumeration of the different patterns for which some Azure services can be used. For some patterns, we also added the K8s-equivalent features/services that may help to address a particular pattern.
- **COMMODITIES:** With this, we cover some transversal requirements, for whatever application.
- **EDA/MESSAGING:** This top-level group is part of our map for the sake of completeness, but we already walked you through it in *Chapter 2, Solution Architecture*.

Let's start with the *DATA* top-level group.

Zooming in on data

In this section, we will focus on the **DATA** top-level group. This will help us to better understand the design patterns. *Figure 5.4* depicts the top-level **DATA** group:

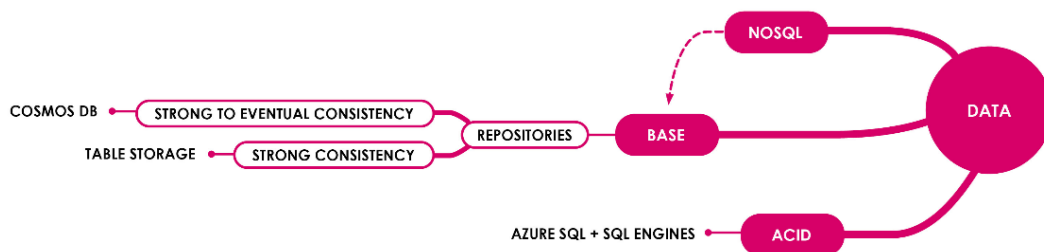


Figure 5.4 – The top-level DATA group

Nowadays, there is a clear trend to default to NoSQL for everything. As an application architect, part of your duties involves stepping back for a second and challenging the actual need of using a NoSQL store. We introduced the notions of BASE and ACID in *Chapter 2, Solution Architecture*. You should review the need for a SQL or NoSQL store, in light of ACID versus BASE, to make sure that NoSQL is indeed appropriate for your use case.

If you opt for **Cosmos DB**, make sure that eventual consistency (which we largely explained in the *Systems of records* section in *Chapter 2, Solution Architecture*) is an acceptable consequence for the business. For example, if you build a content management system that hosts blogs, you can easily use Cosmos DB in full eventual consistency mode, because the number of likes and comments per blog post is not critical information. Nobody really cares whether one visitor sees 12 likes while another one sees 15 likes for the same post. Eventually, they will both see 15, and that is perfectly acceptable. However, if you are selling products online, you would not want two visitors to see different prices for the same product and, eventually, see the same price. You want them to see the correct price directly.

Cosmos DB can, of course, support stronger consistency models, but this is at the cost of write speed, which, in this case, might lead you to ultimately discard Cosmos DB. **Table Storage** is an alternative NoSQL engine with strong consistency. For real big data scenarios, you should instead opt for Cosmos DB, which can scale infinitely if well designed, as opposed to Table Storage, which comes with clear boundaries. Our message here is this: as a rule of thumb, first, always try to plan the consistency model that you need before you choose between a SQL or NoSQL store. Of course, for heavily relational data, you should default to a SQL engine, and **Azure SQL** is probably the best fit in Azure.

Now that we have reminded you of what is important when choosing a database engine, let's discuss where Cosmos DB shines with some of our **CLOUD DESIGN PATTERNS**, that is, our second top-level group.

Zooming in on cloud design patterns

To be able to build cloud and cloud-native apps, you need to understand their underlying design patterns. In this section, we will tackle the most frequently used ones. The design patterns depicted in this map (see *Figure 5.5*) do not uniquely belong to the cloud world because you can also apply most of them to on-premises hosted apps. They just belong to modern development techniques, and, as we explained before, cloud-native applications are modern by design:

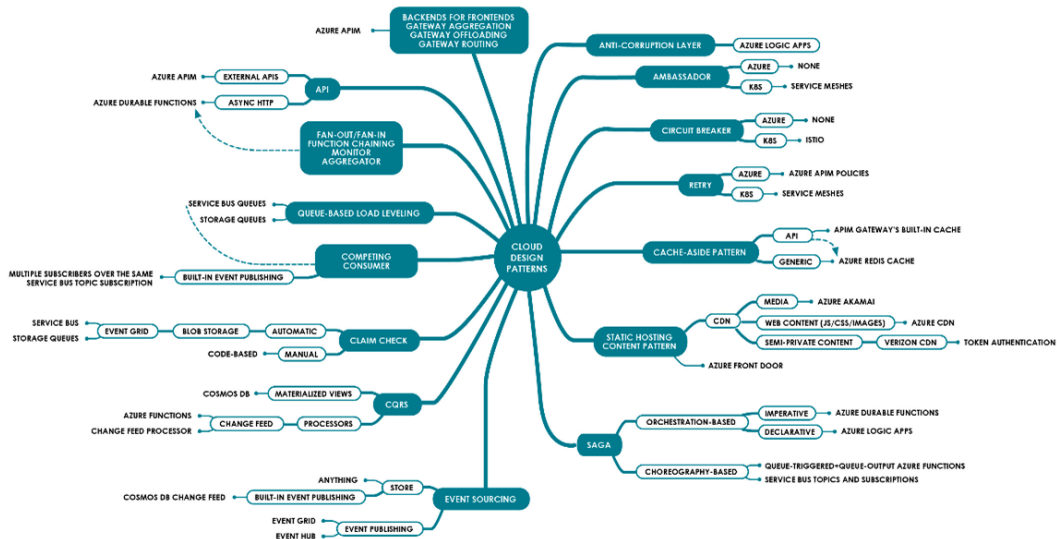


Figure 5.5 – Cloud design patterns

Important note

To see the full **Cloud Design Patterns** map (Figure 5.5), you can download the PDF file here: <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/blob/master/Chapter05/maps/Azure%20Application%20Architect%20-%20Cloud%20Design%20Paterns.pdf>.

In this section, we will only briefly describe some patterns. Sometimes, we'll spend a bit more time to review a pattern deeper, and we'll link it to Azure and/or K8s services. The associations we make between a given pattern and a service is by no means the only possible option, but we just want to demonstrate how you can use some Azure services to your advantage for certain patterns. Let's start with a very popular pattern, namely, **Command Query Responsibility Segregation (CQRS)**.

Using Cosmos DB with CQRS

Figure 5.6 illustrates some Azure options that can help you to deal with the CQRS design pattern:



Figure 5.6 – Azure services and CQRS

In a nutshell, **CQRS** is a pattern that segregates commands (writes) and queries (reads). The rationale behind this is to increase speed and scalability for both write and read operations, and also to reduce the pressure on the datastore by splitting write data from read data into different datastores (or partitions). In this way, potentially heavy read operations do not impact the write performance and vice versa. As you might have guessed already, working with different datastores automatically implies eventual consistency. That is one of the reasons why Cosmos DB and CQRS go well together. Another reason is that CQRS will force you to properly design your Cosmos database. The performance of your Cosmos DB is dramatically impacted by the following factors:

- Logical partitions
- Evenly distributed data
- Cross-partition queries

Cosmos DB can potentially scale indefinitely if the preceding factors are well thought out, and CQRS will help you to keep them under control. Everything is organized around logical partitions, which, in turn, use physical partitions. While we cannot access the physical layer, which is controlled by Microsoft, we can control the logical layer. Logical partitions are created according to the number of partition key values. If we have 10 books in a collection, and we use the ISBN number as the partition key value, we will end up with 10 different logical partitions with one book each. If we have 10 books, but we decide to choose the author as the partition key, we might end up with 5 different partitions.

A unique value will guarantee an even distribution of the data, but it might not be very query efficient. For example, taking the ISBN number as a partition key but creating reports of books per author will lead to systematic cross-partition queries that will affect costs and performance. On the other hand, very unbalanced logical partition sizes will also have an impact on performance. Another factor to consider is the maximum size of a single logical partition, which is, at the time of writing, 20 GB.

With CQRS, you will be forced to think about your command and query needs upfront, which will then help you to design the Cosmos DB collections and partition keys that you would like to use. Your write operations will target one or more collections, while your query needs might be projected using **materialized views**. *Figure 5.7* shows a simplified CQRS implementation with Cosmos DB:

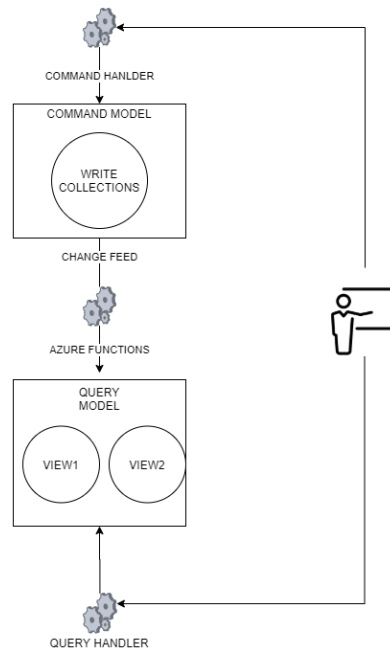


Figure 5.7 – CQRS and materialized views in Cosmos DB

You may have one or more write collections, working with different partition keys, and plenty of different materialized views that satisfy your query needs and that have their own dedicated throughput. To create the materialized views, you may rely on Cosmos DB's built-in change feed, which is a persistent record of changes happening in the collections. The change feed can be consumed in parallel by **Azure Functions**, or any other handler, through the **change feed processor**, to build multiple materialized views from the same initial change.

Materialized views are then consumed by presentation layers through query handlers. Keep in mind that CQRS represents an extra complexity, compared to the traditional way of working with SQL databases. So, again, make sure it is applicable to your use case. CQRS might be achieved completely differently, but Cosmos DB might help in that matter. You can also perfectly apply CQRS with Azure SQL, but this will surely be a bit harder to achieve. Note that, independently of the full pattern implementation, you can easily segregate reads and writes by leveraging Azure SQL's read replica feature. This is by no means CQRS per se, but the physical segregation of reads and writes will help you to realize some of the benefits of CQRS (in terms of performance and scalability) without any effort required on the programming side. You can read more about read replicas at <https://docs.microsoft.com/azure/azure-sql/database/read-scale-out>.

Now, let's look at another pattern that is also sometimes used in conjunction with CQRS, namely, **Event Sourcing**.

Using Cosmos DB with Event Sourcing

Event Sourcing is another popular design pattern. With Event Sourcing, all the events are stored in an event store and published for consumption to build materialized views:



Figure 5.8 – Azure services and the Event Sourcing pattern

So, here again, writes and reads are separated, but the pursued objective differs since the primary purpose of Event Sourcing is the audit trail and extreme scalability. Since events are immutable, we can understand the full history of the data at any point in time. Even better, we can rebuild the actual data state by replaying events from the event store. Cosmos DB's change feed comes in handy because you can use a collection to store the actual events and rely on the change feed to be notified, as well as replay them from any point in time. The change feed persists all the events and restores all of them as they happened. *Figure 5.9* shows how event sourcing can be tackled with Cosmos DB:

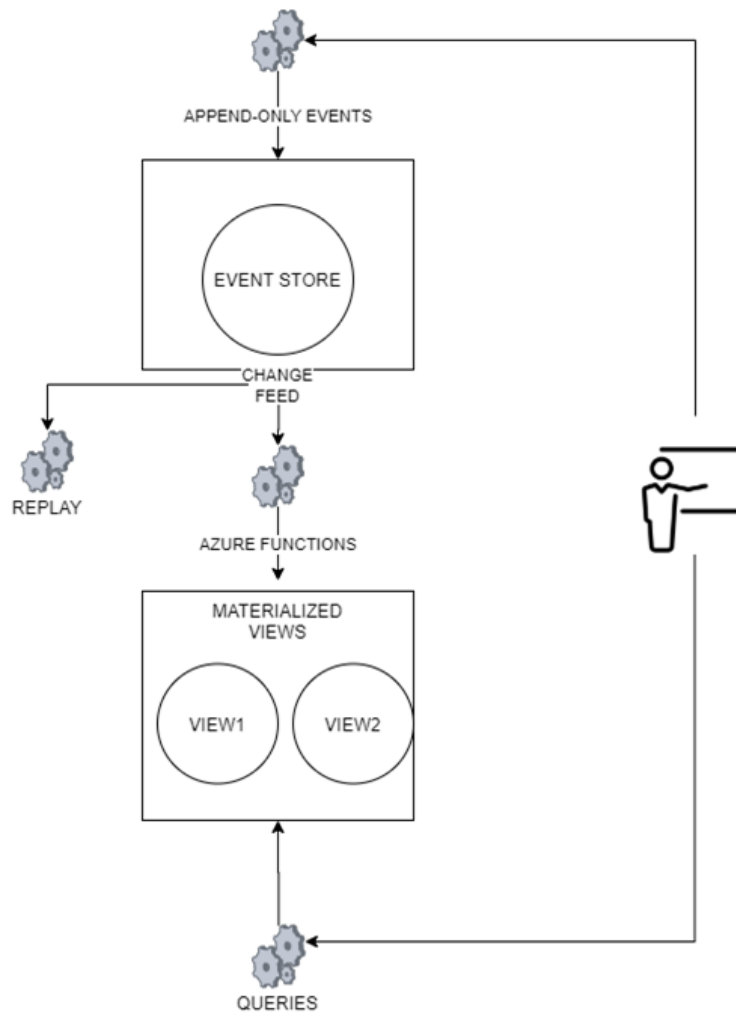


Figure 5.9 – Event Sourcing with Cosmos DB

At first sight, *Figure 5.9* looks very similar to *Figure 5.7*. Indeed, we reuse the same components as before. However, this time, instead of storing data into the write collections, we simply store events as they happened. The change feed is used to fulfill two duties:

- Build the materialized views, as we did with CQRS
- Possibly reconstruct the state of the data by replaying events

To publish events, you may also rely on **Event Hub** and **Event Grid**. These two options may help you to reach out to non-Azure parties. Event Hub is traditionally used for big data, while Event Grid is used for discrete events; however, both work in an Event Sourcing architecture. Choosing one or the other will also depend on the consumer capabilities. To accomplish this, you might have a separate Azure function that publishes the events to either of these services or to both. It's relatively simple to consume Cosmos DB's change feed with Azure Functions in order to build materialized views. You can view a concrete implementation at <https://github.com/Azure-Samples/cosmosdb-materialized-views/tree/master/materialized-view-processor>. Replaying all the events stored in the event store should be done through the change feed processor. Here is an example for illustration purposes. The following method demonstrates how you can replay all the events since the beginning of the event store:

```
private static async Task<ChangeFeedProcessor>
ReplayChangeFeed(CosmosClient cosmosClient)
{
    string databaseName = "packt";
    string sourceContainerName = "events";
    string leaseContainerName = "replay";

    Container leaseContainer = cosmosClient.GetContainer(
        databaseName, leaseContainerName);
    ChangeFeedProcessor changeFeedProcessor =
        cosmosClient.GetContainer(
            databaseName, sourceContainerName)
        .GetChangeFeedProcessorBuilder<Event>(a
            processorName: "replay", HandleChangesAsync)
        .WithInstanceName("replayservice")
        .WithLeaseContainer(leaseContainer)
        .WithStartTime(DateTime.MinValue.ToUniversalTime())
        .Build();
```

```
        await changeFeedProcessor.StartAsync();  
        return changeFeedProcessor;  
    }
```

Here, we tell the processor library that we want to restart from the very beginning using the `.WithStartTime` method. Changes are all sent to the `HandleChangesAsync` method. Here is its implementation:

```
static async Task HandleChanges(  
    IReadOnlyCollection<Event> changes, CancellationToken  
    cancellationToken)  
{  
  
    foreach (Event ev in changes)  
    {  
        Console.WriteLine($"Detected operation for item  
            with id {ev.id}");  
        await Task.Delay(10);  
    }  
}
```

Changes are sent in batches and can be handled. In our example, we simply write the event identifier to the console. This code sample was simply to show you how trivial it can be to inspect the event store.

It is worth mentioning that both Event Sourcing and CQRS are often associated with a broader architectural approach, namely, **Domain-Driven Design (DDD)**. DDD consists of identifying business domains. We will tackle its vocabulary (ubiquitous language) and some bounded contexts later on in the *Developing microservices* section. However, before we get there, let's look at some purely cloud-native patterns.

Dealing with cloud-native patterns

Figure 5.10 shows a few recurrent cloud-native patterns and how Azure or K8s (through Azure AKS) can help tackle them:

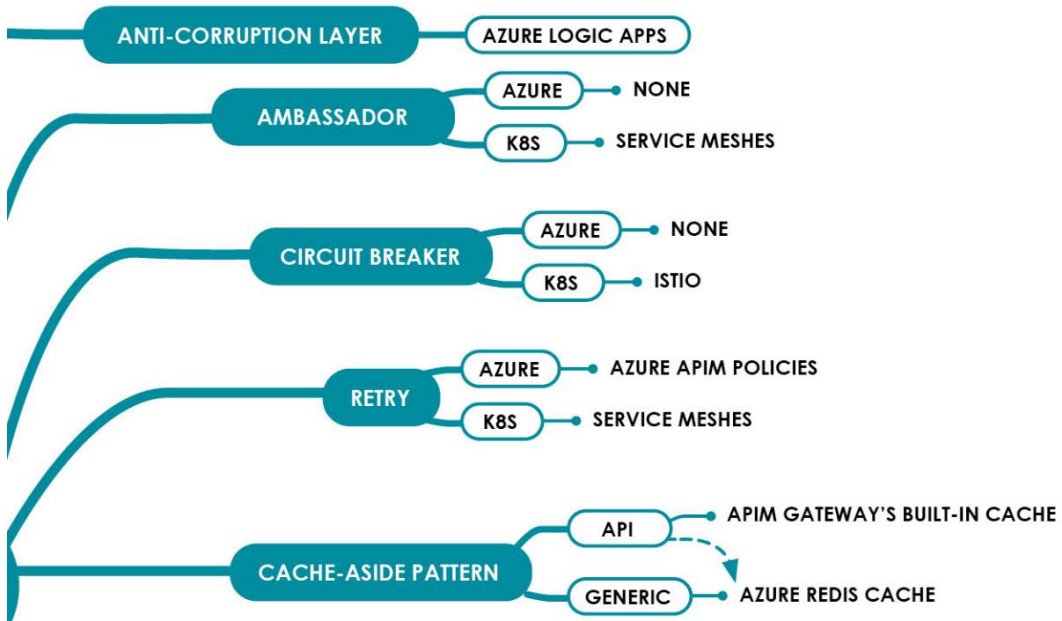


Figure 5.10 – Azure, K8s, and cloud-native design patterns

In DDD, the notion of an **ANTI-CORRUPTION LAYER** refers to the capacity of integrating different systems that do not speak the same language. Hence, there is a risk of corrupting a domain if no specific action is undertaken. This may happen when integrating legacy and newer systems together and, sometimes, even across new systems. Domain corruption could also come from the fact that we are ruled by a broader system that cannot be changed, such as an SaaS platform.

There are numerous reasons why an anti-corruption layer may be required at some point in time. **Azure Logic Apps** can play an interesting role in that matter by applying data mapping and transformations across systems. Azure Logic Apps has hundreds of built-in connectors, which can be extended by custom connectors.

The **Ambassador** and the **Circuit Breaker (CB)** are also cloud-native patterns. It comes as no surprise that Azure has no built-in service to help implement them. The CB pattern is often necessary when dealing with microservices architectures, where services can be quite chatty. Sometimes, these round trips among services may lead to network exhaustion if the retry mechanism is not kept under control. The difference between a simple retry mechanism and a CB is that the CB will stop retrying precisely to prevent network saturation, and it will maintain a state to know when to retry and when not to retry. A direct benefit of implementing it is that your application should be able to survive with the unavailability of some services, forcing you to anticipate service unavailability, from a functional perspective. Azure itself does not have anything, but the **Polly** code library (<https://github.com/App-vNext/Polly/wiki/Circuit-Breaker>) comes to the rescue. In the K8s world, you can count on **Istio** to do the job, since Istio, like other service meshes that we covered in *Chapter 3, Infrastructure Architecture*, implements the Ambassador pattern through a sidecar proxy to perform network-related activities. *Figure 5.11* represents the Ambassador pattern:

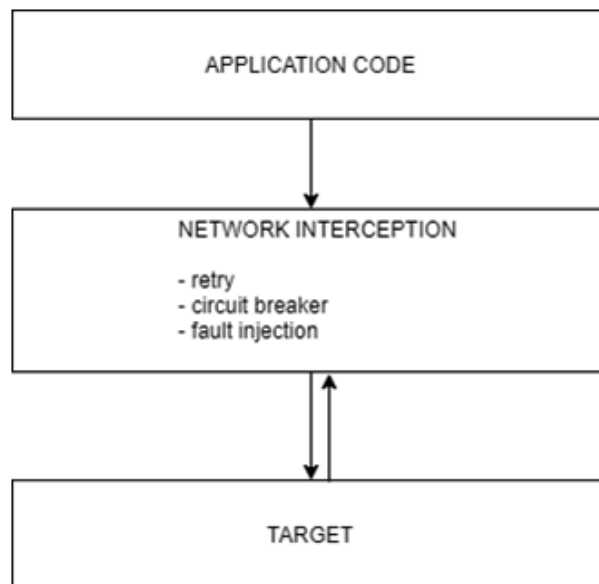


Figure 5.11 – The Ambassador pattern

At the time of writing, only Istio has a true built-in CB, while Linkerd exposes some simpler retry policies.

The **Cache-Aside** pattern could have been put in the **COMMODITIES** top-level group, because many applications require some caching mechanisms. However, the reason why we put it inside the **CLOUD DESIGN PATTERNS** group is to stress, once more, how caching should be implemented in the cloud, and even more in cloud-native apps. The scaling story in the cloud is a scale-out story. This means that we're multiplying the number of instances rather than increasing the CPU and memory (which is scaling up). The Cache-Aside pattern should then be implemented with a distributed cache system, such as **Azure Redis Cache**, to prevent in-process cache and inconsistencies across process instances. *Figure 5.12* is a representation of the Cache-Aside pattern:

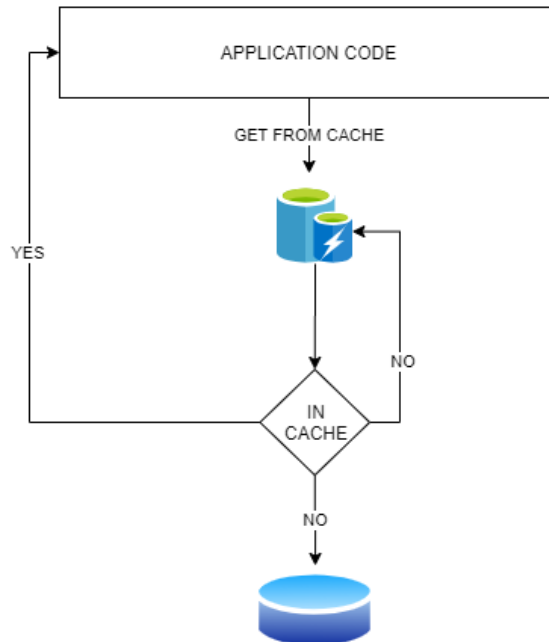


Figure 5.12 – The Cache-Aside pattern

The code checks whether a given value is in the cache or not. If it's not in the cache, it goes to the datastore and updates the cache for future use. If the item is already in the cache, it does not reach out to the datastore. The purpose of the Cache-Aside pattern is to improve performance. Azure Redis Cache is fully managed and allows you to achieve advanced scenarios, such as geo-distributed applications. Let's now explore some API patterns.

Tackling API patterns

APIs are a modern method for integrating systems and exposing some services to client applications. Therefore, it is important to understand some associated patterns, as shown in *Figure 5.13*:

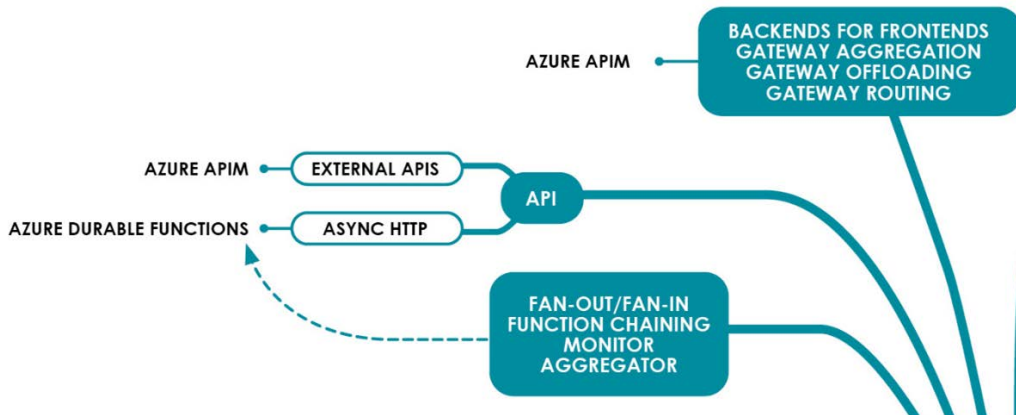


Figure 5.13 – Azure and the API-related patterns

In Azure API Management, the API gateway and its policy engine help to deal with the following gateway-related patterns:

- **Gateway Aggregation:** This pattern facilitates the life of the consumer by aggregating multiple backend sources together. It may also be used as a proxy for some devices with poor (or sometimes poor) network latency, such as mobile and **Internet of Things (IoT)** devices. The idea is to have the API gateway closer to both the consumer and backend services, while letting the client call a single endpoint and allowing the gateway to aggregate multiple calls to the backend on behalf of the client.
- **Gateway Routing:** Here, the purpose is to again make the life of the API consumer easy. This time, we do not aggregate multiple calls to backend services, but we hide those backend services. Our API gateway is an abstraction layer between the client and the backends. The gateway routes a request to the relevant backend according to the incoming HTTP request (for example, the headers, API version, and more).
- **Gateway Offloading:** This pattern consists of offloading cross-cutting concerns to the gateway instead of writing everything in code. Typical duties involve certificate management, authentication, monitoring, and throttling. The idea is to block any illegal request at the level of the gateway instead of letting it flow to the backend service. In the 21st century, it is not possible to handle all these things in code. Note that the gateway could also terminate SSL connections. This can be useful as a proxy to old legacy services that are not always encrypted.

- **Backends for Frontends:** Different frontends may have different needs. For instance, a mobile device might display less information than a web browser on a laptop. Having a single backend service to serve multiple frontends may be challenging at some points since frontend needs might be conflicting. Ideally, *you should implement one backend per frontend*, but you may perform some light transformations at the gateway level according to the calling client through outbound policies. This should remain light so as to avoid setting business logic at the level of the gateway. A typical transformation could be the response conversion from XML to JSON.

Figure 5.14 shows how Azure API Management, which encompasses the API gateway, helps you to implement the aforementioned patterns:

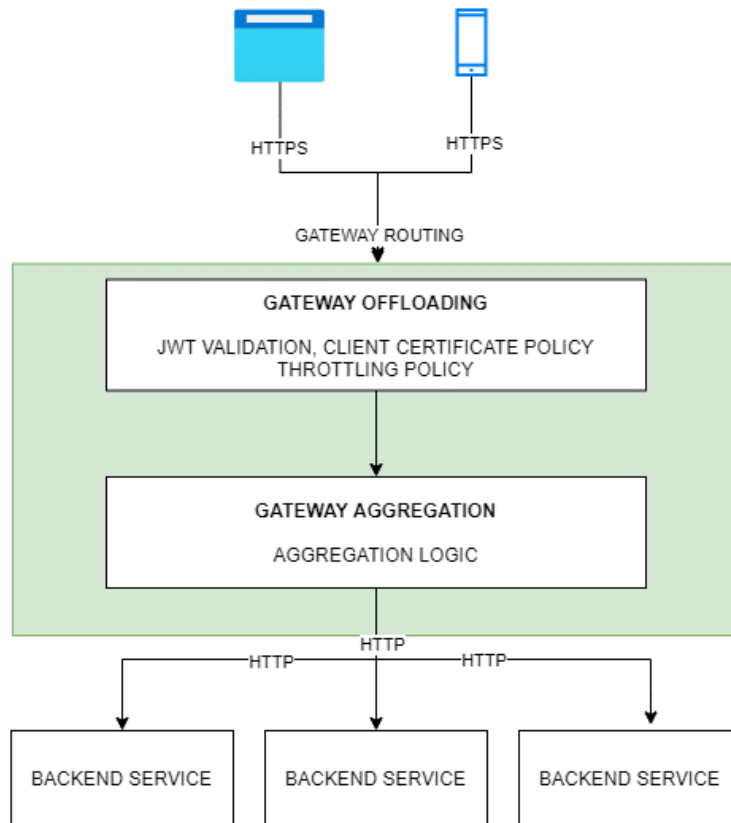


Figure 5.14 – Gateway-related patterns

Needless to say, Azure API Management can also be used to expose APIs to external parties. It is even the primary use case. When exposing functionality to third parties, we want to make sure that we hide internal implementation details. Most of the time, the API gateway sits in front of custom-developed backend services, but it may also proxy other Azure services, such as Azure Functions and Azure Logic Apps. The consumer experience is unified, but the backend implementations may be very diverse.

For long-running operations, you may also leverage **Azure Durable Functions**. They have a built-in support of asynchronous APIs. *Figure 5.15* illustrates what asynchronous APIs are:

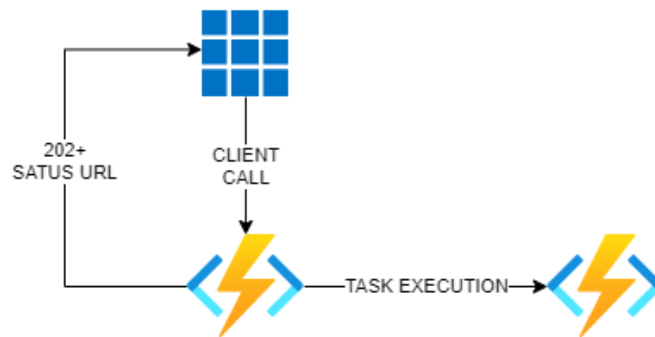


Figure 5.15 – Asynchronous APIs

When the client application calls the function endpoint, it gets a 202 (accepted) response with a URL in the response headers, which can be used to obtain the status. In the meantime, the task execution starts, involving one or more functions. Unlike ordinary functions, the overall process of durable functions is not limited in duration.

Understanding the SAGA pattern

The **SAGA pattern** deals with distributed transactions. This means that there are multiple participants who contribute to the same overall transaction. In the ACID world, a transaction is an atomic group of operations. They all succeed, or they all fail. This works well with monoliths when a single backend writes to a SQL database. With microservices, the segmentation and segregation of duties across services themselves causes the transactions to be distributed by design. Additionally, most of the time, each service has its own datastore, which sometimes does not even support single transactions. The ACID way of working is, therefore, not applicable with true microservices architectures.

Additionally, cloud and cloud-native implementations often rely on serverless systems, such as Logic Apps, that do not even understand the concept of a transaction. So, if you cannot use database-level transactions, you have to rely on a different mechanism, such as the SAGA pattern. Unlike ACID, SAGA cannot simply roll back the whole transaction because of one failed operation, but instead it introduces the concept of compensating transactions. Local transactions committed by local services are already persisted to their own datastore. In the case of a failure of one of the participants, the mechanism of compensating transactions should make them invalidate what was committed.

SAGA is an orchestrator-based or choreography-based pattern. *Figure 5.16* represents the orchestrator pattern:

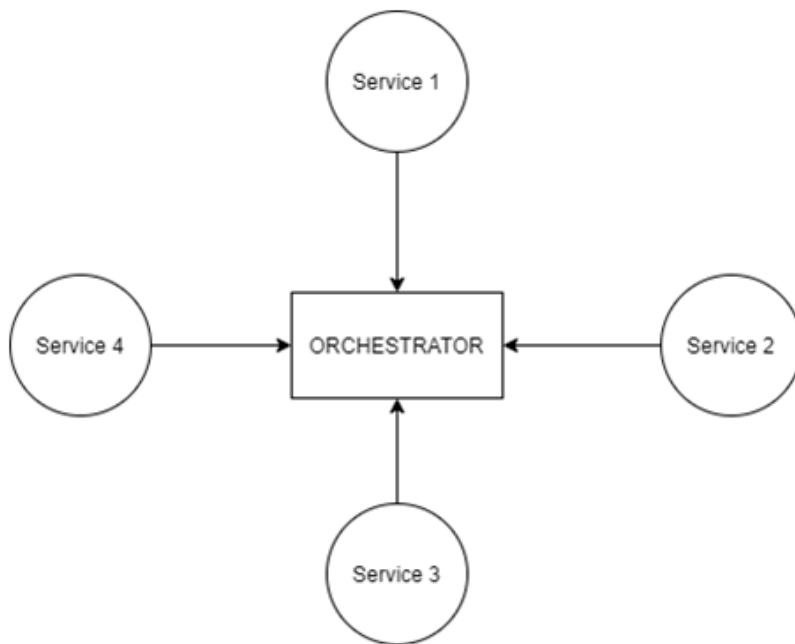


Figure 5.16 – The SAGA orchestrator-based pattern

An orchestrator is the driving seat and orchestrates the different participants that are part of a transaction. Each service is unaware of what other services do and only the orchestrator knows the actual state of the ongoing transaction. If one of the local steps fails, the orchestrator will trigger one or more compensating transactions to invalidate the preceding steps.

In a choreography, as shown in *Figure 5.17*, services are bridged together with a pub/sub mechanism, where each service publishes its outcomes (that is, a success or a failure) that one or more subscribers capture, and, in turn, it publishes events about the outcome of its local activities:

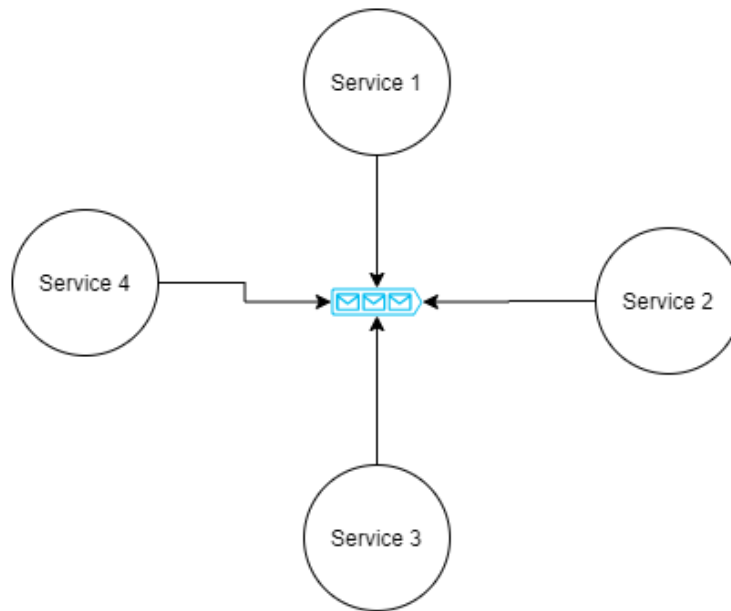


Figure 5.17 – The SAGA choreography-based pattern

If one of the participants fails, it will publish its failure event, which will be captured by the others to invalidate their local transaction. However, the choreography pattern might make it harder to understand the full picture if a single transaction involves many participants.

In Azure, the choreography pattern must use a message broker (such as Azure Service Bus, Azure Event Grid, or Azure Event Hub) to enable participants to publish and receive messages and events.

The orchestrator may use them, but it is not forced to do so. For example, we can use Azure Durable Functions to let it manage its own local state in Azure Storage and coordinate the different participants of a distributed transaction. They can reach back out through HTTP, through a bus, or through any supported Azure Functions binding, among which is Azure Event Hub. Microsoft has published a nice SAGA orchestration example on GitHub: <https://github.com/Azure-Samples/saga-orchestration-serverless>.

Now that we have reviewed the typical cloud design patterns, let's go through a few common requirements.

Understanding the COMMODITIES top-level group

Our **COMMODITIES** top-level group, which is part of our Azure Application Architecture Map, deals with common requirements (see *Figure 5.18*):

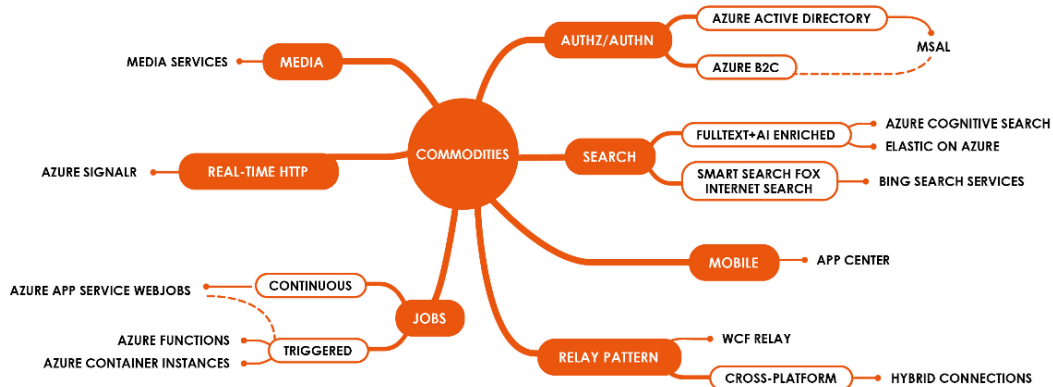


Figure 5.18 – The COMMODITIES top-level group

One of the requirements includes **authentication (AuthN)** and **authorization (AuthZ)**, which, in Azure, often translates to **Azure Active Directory (AAD)** and **Azure B2C**. While AAD is mostly used to authenticate internal collaborators, it can also be used in B2B scenarios. Azure B2C is used to authenticate lambda users (that is, ordinary people like you and me) with their social identities or by letting them create an account in our B2C directory. However, both AAD and Azure B2C can also act as authorization servers and are fully compatible with OpenID Connect. We will explore this further in *Chapter 7, Security Architecture*. Just note for now that as an application architect, you will be interested in **MSAL (Microsoft Authentication Library)**, which is a code library that can be used for all kinds of authentication and authorization scenarios with both AAD and Azure B2C.

Many applications also allow users to search for information. When dealing with a simple use case (for example, building a search UI), calling data services that implement **OData** might be enough. However, as the amount of data grows, the simple OData implementation might not scale accordingly. That is where a fully managed search service, such as **Azure Cognitive Search (ACS)**, comes in handy. ACS lets you define real-time indexes and has built-in AI capabilities such as OCR, translation, text analytics, and more. Azure Cognitive Search integrates with the other cognitive services. For any mobile application, **Visual Studio App Center** is your friend, because it integrates with source code repositories and makes it easy to build and deploy iOS and Android apps whatever source code (for example, Xamarin, React Native, and more) is used.

Background tasks, such as scheduled and triggered jobs, can be handled with Azure Functions. You should always default to functions with one exception (continuous jobs), for which you need to use **Azure App Service WebJobs**.

For hybrid applications (with a part of the application in the cloud and another one on-premises), you should leverage your hybrid infrastructure setup, which we discussed in *Chapter 3, Infrastructure Architecture*. However, if you do not have such a setup, you can always fall back on **Hybrid Connections** and the **Azure Relay pattern**. They both allow you to bridge cloud-based components with on-premises components without the need of establishing private connectivity. Both Hybrid Connections and Azure Relay rely on internet connectivity, and they both require outbound connectivity only. Hybrid Connections come in two flavors, that is, with the Hybrid Connection Manager for databases and as a custom developed endpoint through a modernized Azure Relay approach. Azure Relay used to rely on **Windows Communication Foundation (WCF)**, and, while it is still possible to use WCF, it is not advised for new applications.

For near real-time web applications, you can use **Azure SignalR**, which is simply a SignalR server managed by Microsoft. SignalR is not new in the .NET space and is based on **WebSocket**. The WebSocket API became mainstream with the arrival of HTML5. The purpose of WebSocket is to reuse the same HTTP connection between a browser and a web server across requests. It also uses frames instead of plain HTTP headers, which minimizes verbosity, and, therefore, the footprint of each request. The server needs to be WebSocket compliant, and that is what Azure SignalR does for you.

Now, let's explore some typical messaging and event-driven application patterns.

Developing applications with EDA/messaging-related patterns

Figure 5.19 shows the Azure services that help you to deal with most EDA/messaging-related patterns:

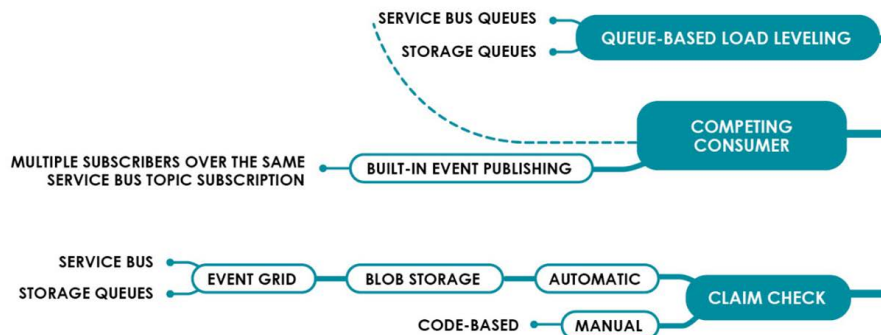


Figure 5.19 – Azure and EDA/messaging patterns

Although we do not embed EDA/messaging-specific details in this map, they support many design patterns, namely, the **Queue-Based Load Leveling**, the **Competing Consumer**, and the **Claim-Check** patterns. Most of these patterns enable asynchronous communication between services (for example, across bounded contexts in a microservices world), and they improve your solution's overall scalability and parallel processing. *Figure 5.20* shows the queue-based load leveling pattern:

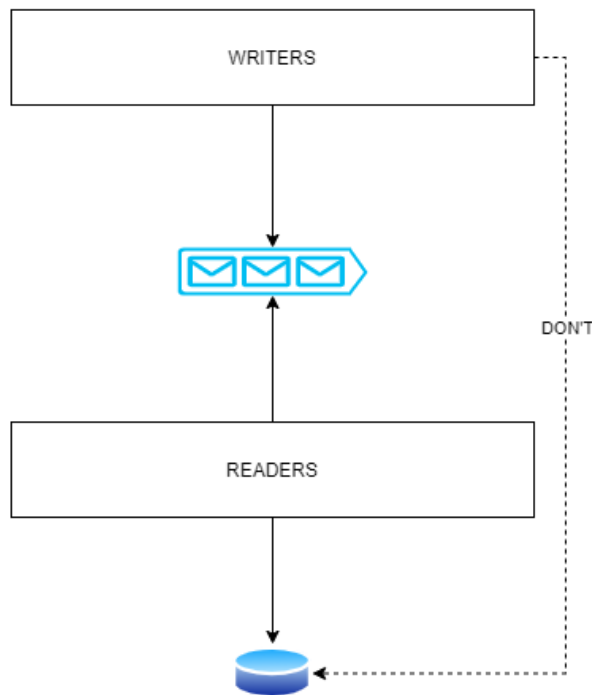


Figure 5.20 – The Queue-Based Load Leveling pattern

Both Azure Service Bus and Storage queues can be used as a messaging layer. The dotted line on the right-hand side (labelled **DON'T**) represents a direct connection between a writer and the database. In highly scalable systems, this should not be done, because the writers might quickly be throttled by the database system and/or exhaust it. The queues serve as a contention mechanism that lets the handlers poll messages at their own pace, in an asynchronous way. The Competing Consumer pattern is just a variant that makes use of multiple handlers against the same queue, in order to increase the speed of message handling. Note that nowadays, it is more common to have pub/sub implementations with topics and subscriptions instead of mere queues.

The **Claim-Check** pattern is also a variant of the Queue-Based Load Leveling pattern and is used when the size of a message is too large. For example, the maximum size of a message in Azure Service Bus is 1 MB for the Premium tier. In such a case, instead of sending the actual message payload to the bus, you place it in Blob storage (or any other datastore) and queue a claim-check as a message to the bus. The receiver can then pull the payload from the Blob storage.

Now that we have browsed some common patterns, and more specifically event and message-based patterns, let's go through a more concrete example of using EDA to our advantage.

Exploring EDAs

In this section, we will dive into the EDA world in a more concrete way. Our objective is to make you familiar with some of the Azure services that help build EDA solutions. By the end of this section, you will understand the basics of Azure Service Bus, Azure Event Hub, and real-time processing with Stream Analytics.

In *Chapter 2, Solution Architecture*, we described and explained the EDA map, as shown in *Figure 5.21*:

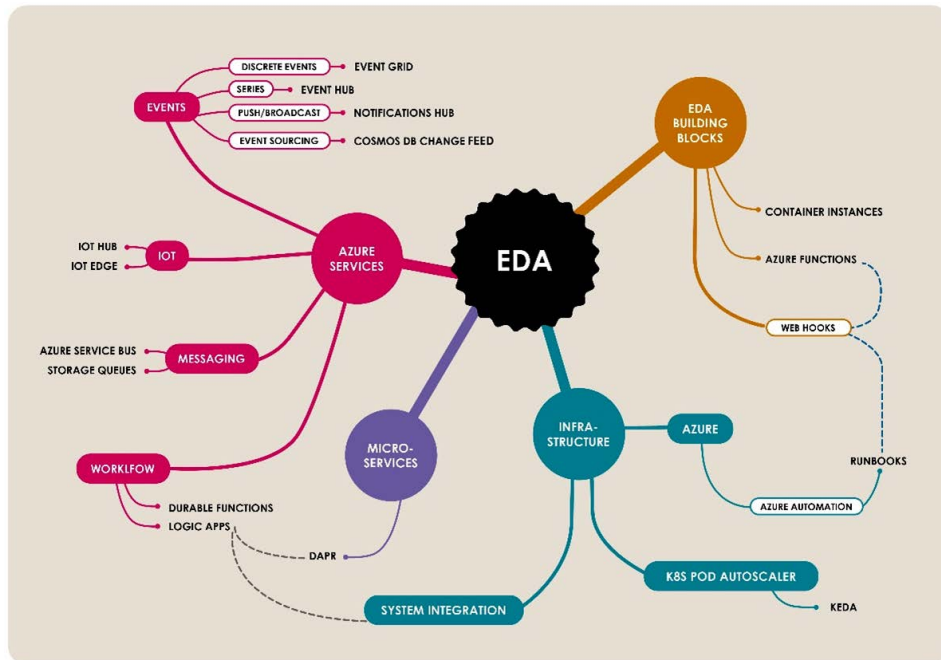


Figure 5.21 – The EDA map

Instead of re-explaining what we have already discussed, let's add some messaging and event capabilities to the application that we developed in *Chapter 2, Solution Architecture*, in the *Solution architecture use case* section. As a reminder, the initial scenario was the following:

Contoso needs a configurable workflow tool that allows you to orchestrate multiple resource-intensive tasks. Each task must launch large datasets to perform in-memory calculations. For some reason, the datasets cannot be grouped into smaller pieces, which means that memory contention could quickly become an issue under a high load. A single task may take between a few minutes to an hour to complete. Workflows are completely unattended (that is, there is no human interaction) and asynchronous. The business needs a way to check the workflow status and to be notified upon completion. Additionally, the solution must be portable. Contoso's main technology stack is .NET Core. Of course, this should have been done beforehand, and there is not much budget allocated to the project.

We ended up building the solution that's shown in *Figure 5.22*:

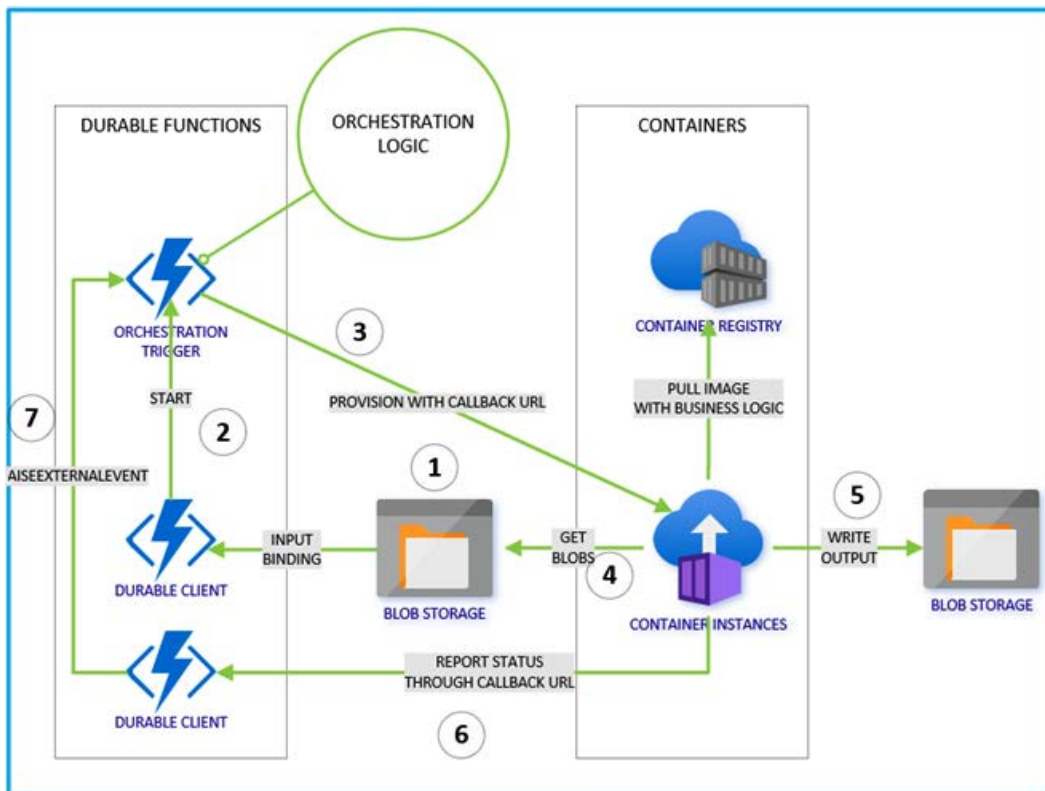


Figure 5.22 – Reference architecture for our solution architecture use case

It appears that Contoso was a bit too quick when launching their workflow system. They realized that it is hard for them to have complete oversight of the ongoing orchestrations; they would like to have a real-time status view. They can easily check the status of a given orchestration, but they do not want to do that one by one. They also noticed that they focused too much on the larger datasets and forgot that they also had to deal with smaller datasets. While the current solution is perfectly capable of handling the smaller ones, users complained about the startup delay encountered by the container instances. So, they urged both the solutions architect and the application architect to solve these problems. Our scenario remains unchanged, except for the following:

- We must have global oversight of the ongoing orchestrations.
- The ACI startup delay is not acceptable for smaller datasets, so we have to find an alternative.

Nevertheless, our main goals remain unchanged, as we still need to have a cost-friendly solution, while also dealing with large datasets (that is, the majority of our datasets). In light of these new requirements, the architects come to the following conclusions:

- The ACIs are still the best option for the larger datasets and are cost friendly.
- Working with pre-provisioned smaller ACIs will prevent the startup delay you encounter when you provision ACIs dynamically. Of course, pre-provisioned ACIs will impact costs.
- The orchestrator and the ACIs are too tightly coupled. Since the system will rely on both dynamic and pre-provisioned ACIs, you need something in between to coordinate the orchestrator steps and the ACI activity. Moreover, we need to find a way to route small volumes to pre-provisioned ACIs while still creating dynamic ACIs for the larger ones. A message broker, such as **Azure Service Bus**, should be added to the mix.
- We already have a way to check the individual orchestration status, but it is reactive and not near-real time. Therefore, we need a way to publish the orchestration status, so we will use **Azure Event Hub**. However, we also need something to pick up those events and reflect them onto a monitoring dashboard. So, we will add **Azure Stream Analytics** to the mix, as well as real-time **Power BI Dashboards**.

Here, we end up with a few extra services, and that will probably also have an impact on our code. Remember that in the previous design, all ACIs were provisioned dynamically and destroyed after use. Now, we will have to deal with pre-provisioned ACIs, meaning that our code will keep running all the time. In such a situation, we want to make sure that we deal correctly with thread safety and memory leaks to prevent crashes of our pre-provisioned ACIs. The fact that we were only working with temporary ACIs could have hidden some code-related issues of that nature. Moreover, by adding Service Bus in the middle, our retry and timeout mechanisms should be reviewed. The application architect reworked the previous solution diagram and delivered something similar to *Figure 5.23*:

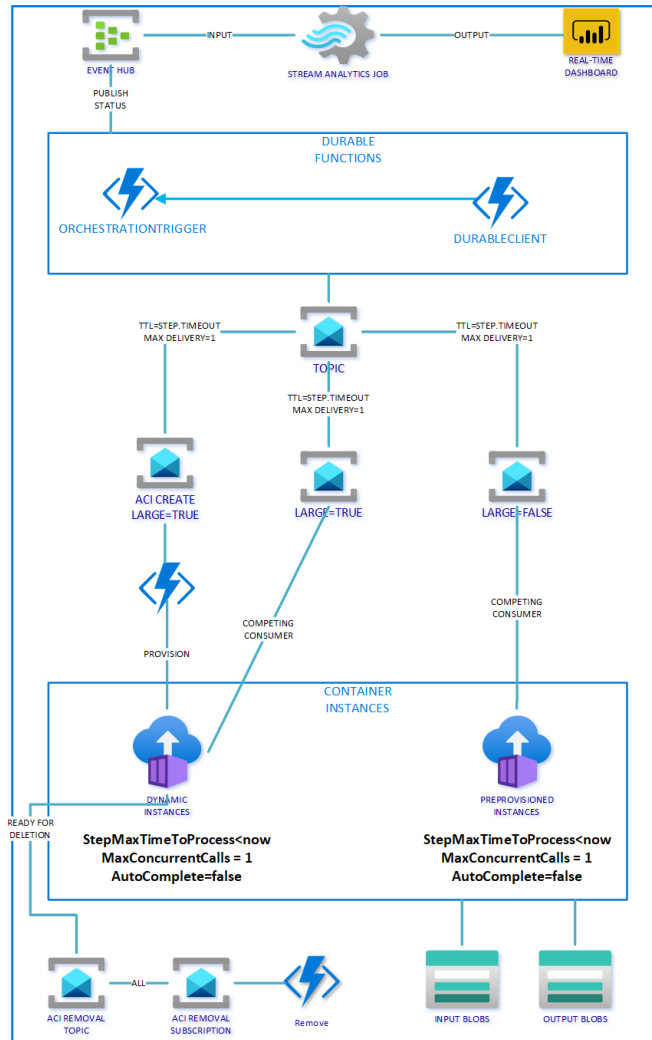


Figure 5.23 – The revisited Contoso architecture

This diagram, as well as the other diagrams, are available as a Visio document on GitHub at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/blob/master/Chapter05/diagrams/diagrams.vsd>.

The flow is as follows:

- For each orchestration step, we determine whether the step targets a small or large dataset. We queue a message to a **Service Bus topic** with the metadata property set accordingly.
- We have three separate **Service Bus subscriptions** for our topic:
 - a) The first subscription only receives messages for which the large property was set to true (large datasets). When received, it provisions a new ACI.
 - b) The second subscription also receives all the messages for which large was set to true. The newly provisioned ACI receives its subscription details as an environment variable.
 - c) The third subscription receives all the messages for which large was set to false (that is, the small datasets). This will feed the pre-provisioned ACIs.
- All ACIs, including any dynamic ones, will be in a **competing consumers** pattern, meaning that the first one to detect the message will pick it up, lock it (that is, hide it from others), and dequeue it upon successful completion. Under a high load, even the dynamically provisioned ACIs may handle a message other than the one they were initially provisioned for. The order of the steps is ensured by the orchestrator, so we do not have to use **Service Bus sessions**.
- As before, dynamic ACIs have to be destroyed after use (for cost reasons). So, they will queue a message to a specific topic when they are ready for deletion. Specific logic must be ensured so that the ACI is not destroyed while it is processing a message. A specific ACI removal subscription will be listening to all ACIs that are to be destroyed, and this will be handled by an Azure function through the **Azure Resource Manager API** (which we saw in *Chapter 4, Infrastructure Deployment*).
- Upon completion of each step, whether successful or failed, the orchestrator will publish the step outcome to **Azure Event Hub**. **Azure Stream Analytics** will pick it up and push it to a real-time **Power BI Dashboard**.

- A number of extra timing constraints must also be considered. We need to make sure that we set the **time-to-live (TTL)** of the Service Bus message to the value of the step timeout in order to let the bus dequeue outdated messages. The message body will also contain the step timeout information. For the sake of precaution, every message handler will double-check that the message it reads is not expired yet. (Sometimes, it can be a question of milliseconds.) Even more importantly, the message handler must make sure that it recalculates a new timeout before it handles the message. Indeed, an orchestration step might time-out in 3 minutes, but by the time we process it, we might go over the initial 3 minutes, which would lead us to a weird situation where the orchestrator would time out while the step was being handled. To avoid this situation, the handler needs to recalculate whether or not it has enough time to process the message before the orchestrator timeout kicks in.
- Like before, Blob storage accounts are used as input/output for our ACI activity.

Figure 5.24 is a concise view of this flow:

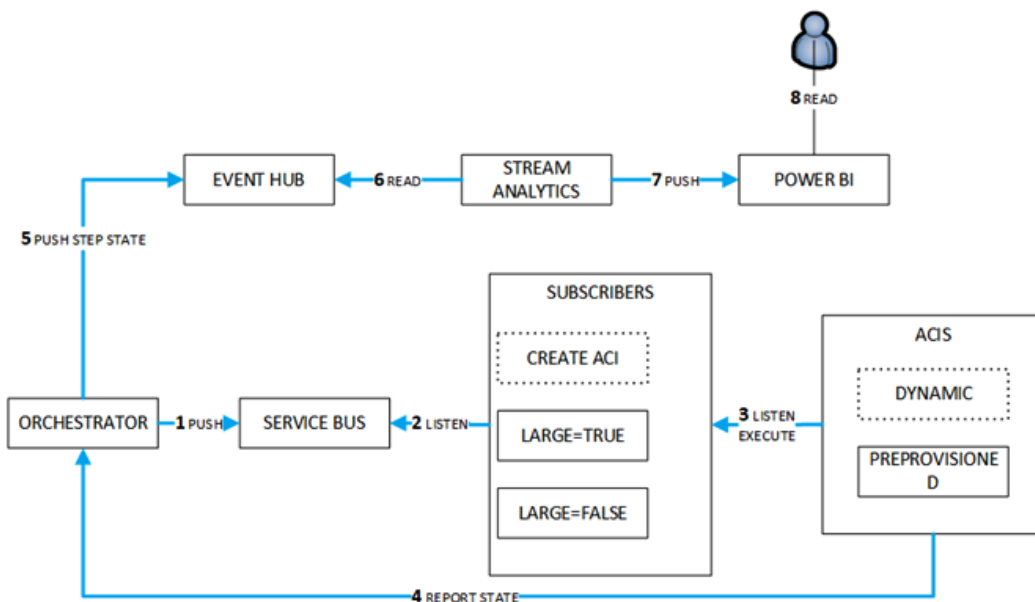


Figure 5.24 – Contoso orchestration flow

Notice the numbers in the diagram that indicate the sequence.

For the sake of brevity, we are not going to go through all of the steps to get this up and running because it is beyond the scope of the book. In the next subsections, we will only highlight the important changes, compared to the previous solution. Next, let's take a look at the Service Bus configuration.

Inspecting the Azure Service Bus configuration

Figure 5.25 shows our **datasets** topic, which receives messages directly from the orchestrator:

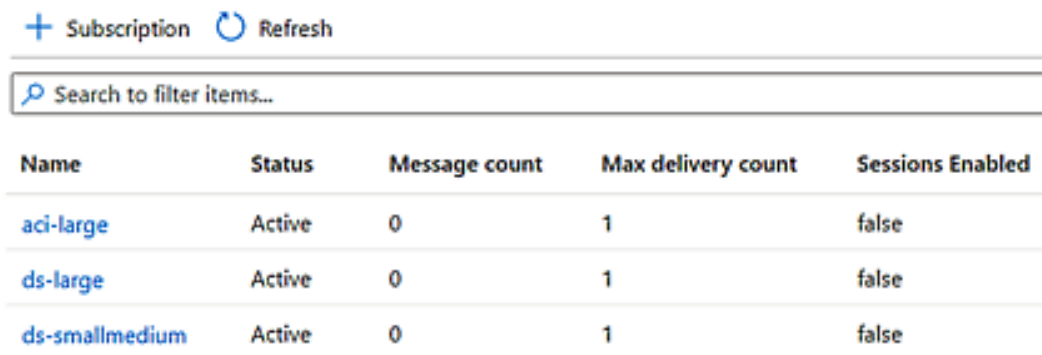


The screenshot shows the Azure Service Bus interface for a topic named 'datasets'. At the top, there are buttons for '+ Topic' and 'Refresh'. Below that is a search bar with the placeholder text 'Search to filter items...'. The main content is a table with the following data:

Name	Status	Max size	Subscription count
datasets	Active	1024 MB	3

Figure 5.25 – Service Bus topic

There's nothing special about the topic; it is just a regular topic. It has three subscriptions (as shown in Figure 5.26):



The screenshot shows the Azure Service Bus interface for the subscriptions of the 'datasets' topic. At the top, there are buttons for '+ Subscription' and 'Refresh'. Below that is a search bar with the placeholder text 'Search to filter items...'. The main content is a table with the following data:

Name	Status	Message count	Max delivery count	Sessions Enabled
aci-large	Active	0	1	false
ds-large	Active	0	1	false
ds-smallmedium	Active	0	1	false

Figure 5.26 – Topic subscriptions

Remember that these subscriptions have some filters applied. Figure 5.27 shows the **aci-large** subscription, which is there to provision dynamic ACIs for large datasets:

The screenshot displays the configuration for the 'aci-large' subscription in Azure Service Bus. At the top, there are several key settings:

- Max delivery count:** 1 (change)
- Message time to live:** 10675199 DAYS (change)
- Message lock duration:** 5 MINUTES (change)
- Auto-delete:** NEVER (change)

Below these are message counts for various categories, all currently at 0:

- Active message count: 0 MESSAGES
- Dead-letter message count: 0 MESSAGES
- Transfer message count: 0 MESSAGES
- Scheduled message count: 0 MESSAGES
- Transfer message count: 0 MESSAGES

The 'FILTERS' section shows a table with one filter:

Name	Filter Type
aci-large	SqlFilter

To the right, the configuration for the 'aci-large' filter is shown:

- Filter Type:** SQL Filter (selected), Correlation Filter
- Filter expression:** 1 large = true

Figure 5.27 – Configuration of the aci-large subscription

As you can see, the SQL filter, `large=true`, is set. This means that Azure Service Bus will send a copy of each message for which `large` was set to `true`. There are a few other interesting parameters here, as follows:

- **Max delivery count:** This is set to `1`. It means that Azure Service Bus will only try to deliver a message once. It is a question of choice here. Either we let Service Bus retry for us, or we let the orchestrator retry. With the max delivery count set to `1`, we let the orchestrator retry it. The rationale here is to put the orchestrator in the driver seat, as it is ultimately responsible of the overall orchestration.
- **Message time to live:** This is set to a very improbable default value. Remember that we want the TTL to be equal to the step timeout, which is unknown at the subscription creation time. Rest assured, we can specify the TTL while queuing a message.
- **Message lock duration:** Here, we specified 5 minutes, which should be more than enough time to provision an ACI. 5 minutes is the maximum duration that we can specify. However, the Service Bus SDK offers some ways to renew the lock, should our operation last longer.
- **Auto-delete:** This is set to `NEVER`. We prefer to **dead-letter** every expired or invalid message.

Azure Service Bus, just like many message brokers, has this notion of **dead-letter queue (DLQ)**. The DLQ is particularly useful because every message ending there is the symptom of a problem. A message could be malformed, expired, or cause filter exceptions. It is important to keep an eye on the DLQ to track down issues. Note that Azure Service Bus configuration can be done entirely programmatically, thanks to the `ManagementClient` class. The following code shows how to create our three subscriptions through code, from a console program, using this class. Note that this code is just for illustration purposes, which is why we do not add a dependency injection (and so on) into the mix. First, the following is our main method:

```
static async Task Main(string[] args)
{
    const string Topic = "datasets";
    ManagementClient mgt = new ManagementClient(
        "connection string");
    try
    {
        await CreateSubscription(mgt, Topic, "aci-large",
            5, new SqlFilter("large = true"));
        await CreateSubscription(mgt, Topic,
            "ds-smallmedium", 5, new SqlFilter(
                "large = false"));
        await CreateSubscription(mgt, Topic, "ds-large",
            5, new SqlFilter("large = true"));
    }
    catch(Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        await mgt.CloseAsync();
    }
}
```

We use the `ManagementClient` class from the Service Bus .NET Core SDK, and we call the `CreateSubscription` method three times. Its implementation is as follows:

```
static async Task CreateSubscription
( ManagementClient mgt,
  string Topic,
  string SubscriptionName,
  int LockDuration,
  SqlFilter filter=null){
  var sub=await mgt.CreateSubscriptionAsync(
    new SubscriptionDescription(Topic, SubscriptionName),
    (filter != null) ?
      new RuleDescription{
        Name = string.Concat(SubscriptionName,"-filter"),
        Filter = filter
      }:null
  );
  sub.MaxDeliveryCount = 1;
  sub.LockDuration = new TimeSpan(0, 5, 0);
  await mgt.UpdateSubscriptionAsync(sub);
}
```

This method simply adds the new subscription to our Service Bus instance, with some default values and the filter rule, if any.

Our orchestrator will publish messages to the topic with the following activity function:

```
[FunctionName(nameof(PublishMessage))]
[return: ServiceBus("datasets", Connection =
  "ServiceBusConnection")]
public static Task<Message> PublishMessage(
  [ActivityTrigger] IDurableActivityContext context,
  ILogger log)
{
  var payload = context.GetInput<
    OrchestrationStepPayload>();

  var message = new Message
```

```

{
    Body = Encoding.UTF8.GetBytes(
        JsonConvert.SerializeObject(payload)),
    ContentType = @"application/json"
};
message.UserProperties.Add(@"large", payload.large);
return Task.FromResult<Message>(message);
}

```

Notice how we promote the `large` attribute onto which subscribers apply a filter. Our function has a return of type `ServiceBus`, for which we provide both the topic and the connection.

Now that our Service Bus plumbing is ready, we also need to add an Azure Event Hub, a PowerBI dashboard, and a Stream Analytics job in the middle.

Adding the other components to the mix

Since we are only interested in the changes that we added to the previously developed solution, we are not going to detail all of the steps. Let's only explore the relationships between the different components. First, we need an event hub. For this, we simply create an instance of Azure Event Hub. Remember that the orchestrator publishes all its orchestration events to the Hub. It does so, thanks to the following activity function:

```

[FunctionName(nameof(PublishEvent))]
public static async Task PublishEvent(
    [ActivityTrigger] IDurableActivityContext context,
    [EventHub("acihub", Connection = "EventHubConnection")]
    IAsyncCollector<string> outputEvents,
    ILogger log)
{
    var data = context.GetInput<OrchestrationData>();
    await outputEvents.AddAsync(
        JsonConvert.SerializeObject(data));
}

```

Our data object contains the step name and its status. We use an output function binding of the `EventHub` type.

The service (which will push those events to Power BI) is an instance of Azure Stream Analytics. We define the event hub as its input and two different Power BI datasets as the outputs. The job will run the query that's shown in *Figure 5.28*:

```

▶ Test query  📄 Save query  ✕ Discard changes
1  SELECT
2  | *,CONCAT ( ParentOrchestrationId, '-', StepName) AS OrchestrationStep
3  INTO
4  | [aci]
5  FROM
6  | [orchestration]
7
8  SELECT
9  | *,CONCAT ( ParentOrchestrationId, '-', StepName) AS OrchestrationStep
10 INTO
11 | [failed]
12 FROM
13 | [orchestration]
14 | WHERE OrchestrationStatus = -1 or StepStatus = -1

```

Figure 5.28 – Stream Analytics job query

Our goal is to stream all of the events as they fly on the hub, and then we output them to an all-in dataset and another dataset that only shows the failed steps/orchestrations. By doing this, we end up with a near real-time monitoring dashboard, as shown in *Figure 5.29*:

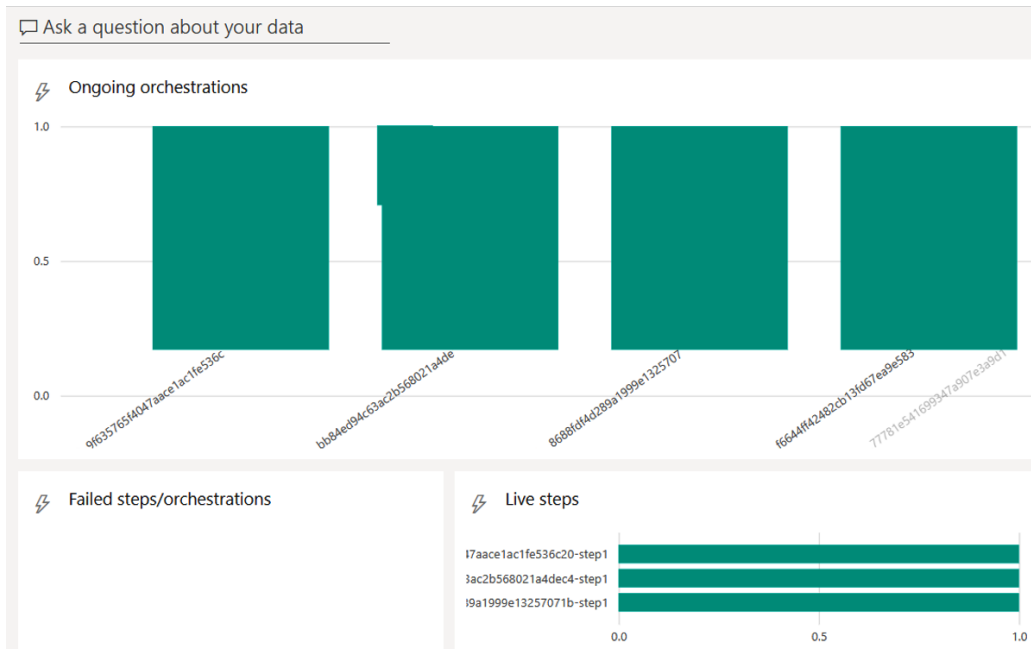


Figure 5.29 – Near real-time monitoring dashboard

The dashboard reflects what is happening at the level of the orchestrator, and it has a specific view of the failed steps (it shows as empty in our dashboard). Let's now recap what we learned from this revamped solution. In *Chapter 6, Data Architecture*, we will walk you through a basic scenario that involves Stream Analytics and Power BI.

What we intended to demonstrate is that when you leverage the ecosystem, in this case a bunch of Azure services, your solution tends to be less code-centric. With the preceding example, the only real application code sits in the ACIs where the business logic lies. All the other services we added to the mix play a crucial role, and most of the work consists of configuring and assembling the services properly. Let's now leverage a different ecosystem for a microservices architecture example.

Developing microservices

Our objective in this section is to give you a glimpse of developing cloud-native applications. Microservices are 100% cloud-native par excellence, hence the reason why we use them to illustrate our purpose. We will not cover the entire spectrum of microservices, because the topic deserves an entire book on its own. We assume that you know already what microservices are, and we will focus on some technical bits only.

In *Chapter 3, Infrastructure Design*, we reviewed the different infrastructure options at our disposal to host microservices. Our conclusion was that, at the time of writing, in Azure, AKS is the most suitable choice. We stressed the fact that service meshes play a great role in terms of the observability, security, deployment, and resilience of a microservices architecture. We also highlighted some AKS-killing features, such as self-healing, cluster, and pod auto-scaling, to name a few. All of these infrastructure features help to build resilient solutions while not reinventing the wheel in the code. Microservices are often part of a broader DDD approach, which we will not depict in this book. However, to make a long story short, we ultimately end up with bounded contexts. Eric Evans, the pope of DDD, defines a bounded context as follows:

"A bounded context is a defined part of software where particular terms, definitions, and rules apply in a consistent way."

Considering that a bounded context *is* a microservice is oversimplifying things, but we can at least say that the physical segregation of services helps to set up clear boundaries between the different contexts.

If you want to know more about DDD and Eric Evans, you can refer to the following Packt page: <https://hub.packtpub.com/eric-evans-at-domain-driven-design-europe-2019-explains-the-different-bounded-context-types-and-their-relation-with-microservices>.

Let's focus on the technical aspects. In a nutshell, microservices tend to rely on the following patterns and technologies:

- **Sidecar:** The Sidecar pattern is one of the most-used patterns in the container world. Service meshes themselves leverage this pattern, just like most ecosystem solutions do.
- **PUB/SUB:** These establish asynchronous communication between services.
- **gRPC and REST HTTP/2-enabled communications:** These speed up direct communications across services.
- **API gateways:** These expose some services to the outside world.

Let's now inspect a pure code-related ecosystem add-on, namely, **Dapr**, which we introduced in *Chapter 2, Solution Architecture*. Dapr can be used with all of the patterns and technologies listed in the preceding list.

Using Dapr for microservices

Dapr comes in very handy when dealing with microservices. As a reminder, Dapr is an application runtime that helps you to work with distributed components. A first release candidate was rolled out in November 2020. Dapr has a great affinity with microservices and K8s. Dapr's value proposal is to be an abstraction layer between the application code and the underlying systems the code interacts with. It totally decouples the code from its targets and allows you to seamlessly substitute a target for another, without changing a single line of code.

Dapr deals natively with the technologies and patterns listed in the previous section. Dapr also works with components, as shown in *Figure 5.30*:

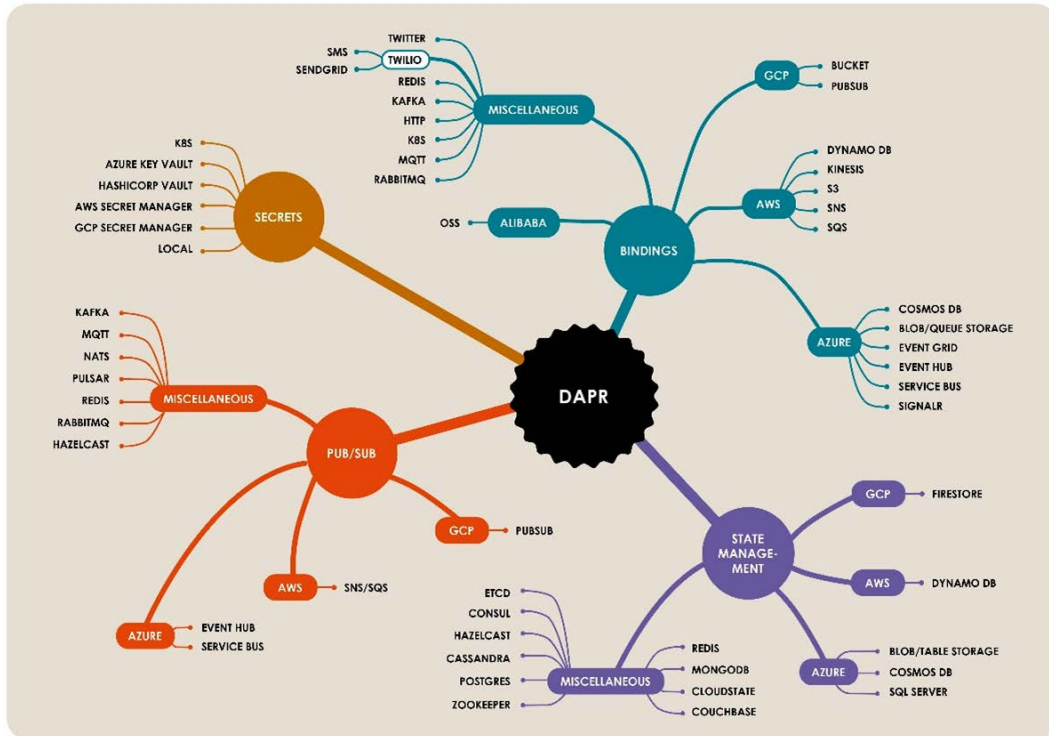


Figure 5.30 – Dapr components

These **components**, as illustrated in *Figure 5.30*, are of the following types:

- **SECRETS:** Many different secret stores are supported by Dapr. Referencing a secret in Dapr is nothing more than making an HTTP (or gRPC) query to `http://localhost:3500/v1.0/secrets/{component-name}/{secret-name}`.
- **PUB/SUB:** Dapr supports Azure Event Hub and Service Bus, but also Redis, RabbitMQ, and more. Publishing a message to a Service Bus topic with Dapr is nothing more than calling `http://localhost:3500/v1.0/publish/{component-name}/{topic}`, while subscribing is achieved with `http://localhost:3500/v1.0/publish/{component-name}/{topic}`.
- **BINDINGS:** Dapr can be bound to many different stores, including Cosmos DB and Azure Storage, to name a few. As usual, a simple query to `http://localhost:3500/v1.0/bindings/{bind-name}` is enough to bind to one of the supported stores.

- **STATE:** Dapr can write key/value pairs to all of the supported state stores. This time you need to target `http://localhost:3500/v1.0/state/{component-name}/{key-name}`.

An extra endpoint is provided by Dapr, but it is not represented by components. Therefore, it is not in our map. This endpoint is as follows:

- **INVOKE/{APPLICATION}/METHOD/{METHOD}:** This is used to directly invoke a service from another service.
- **INVOKE/WORKFLOWS/METHOD/{WORKFLOW}:** This is used to invoke a workflow. Dapr integrates with Azure Logic Apps to execute self-hosted workflows while leveraging the power of Logic Apps.

Now, let's introduce the concept of Dapr components.

Understanding Dapr components

Dapr makes use of components to abstract away the various data and message stores that client applications interact with. *Figure 5.31* shows how the Dapr sidecar container reads the component configuration to eventually target the actual physical stores, such as the ones represented in our map (see *Figure 5.30*):

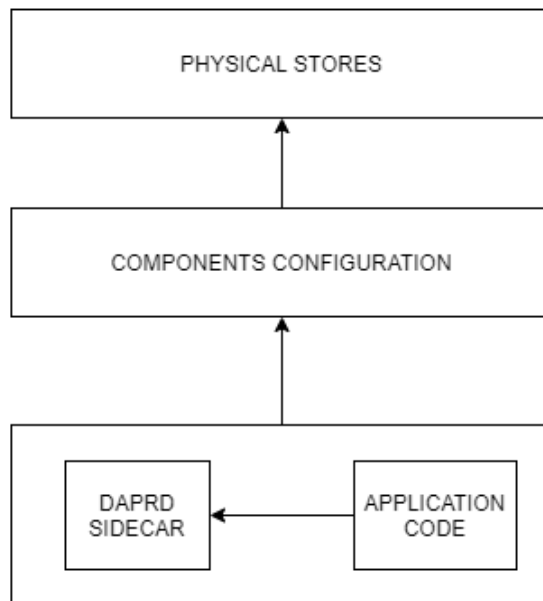


Figure 5.31 – Dapr components

The application code does not know that it is talking to Azure Storage or Amazon S3. It just talks to one of the endpoints that we discussed in the previous section. It is up to the sidecar container, named **Daprd**, to find the relevant component. The benefit of this approach is a great simplification of the application code, but also a full decoupling of the application code and the stores it interacts with. Thanks to this architecture, we can easily switch physical stores by just changing the component configuration, without impacting the application code. A component looks like the following:

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: {name}
  namespace: {K8s namespace}
spec:
  type: {component-type}
  metadata:
    - {component-specific}
```

The K8s custom resource definition is **Component**, and the metadata varies according to the type of component. Once deployed to AKS, Dapr will detect its presence upon starting the sidecar container. Note that for experimental purposes, you can also run Dapr in standalone mode, without K8s, and that's what we are going to do next. Now that we have a better understanding of the components, let's find out how to get started with the application code.

Getting started with Dapr SDKs

The application code only calls the various Dapr endpoints that correspond to the Dapr components, and that is why Dapr is compatible with *any* programming language. However, to keep developers in their comfort zone, Dapr also delivers SDKs. For .NET Core, Dapr already ships with quite a few SDKs (see *Figure 5.32*):

The screenshot shows the NuGet package manager interface. At the top, there are tabs for 'Browse', 'Installed', and 'Updates'. Below the tabs is a search bar containing the text 'Dapr'. To the right of the search bar are icons for a dropdown menu, a refresh button, and a checkbox labeled 'Include prerelease' which is checked. Below the search bar, a list of packages is displayed. Each package entry includes a logo, the package name, a checkmark, the author, the number of downloads, and the version number. A description for each package is provided below the name.

Package Name	Author	Downloads	Version
Dapr.Client	dapr.io	28.6K	v0.12.0-preview01
Dapr.AspNetCore	dapr.io	24.3K	v0.12.0-preview01
Dapr.actors	dapr.io	12.7K	v0.12.0-preview01
Dapr.actors.AspNetCore	dapr.io	8.74K	v0.12.0-preview01
Dapr.WebSockets	Reuben Bond	14.8K	v0.2.7
Dapr.AzureFunctions.Extension	dapr.io	846	v0.10.0-preview01

Figure 5.32 – Dapr .NET Core SDKs

As you can see, there are multiple packages, including an integration with Azure Functions. In the next section, we are going to demonstrate a scenario that should help you to grasp the overall concept.

Looking at our scenario

Before we look at the details, let's look at the broader picture. *Figure 5.33* shows you the application that we are going to build:

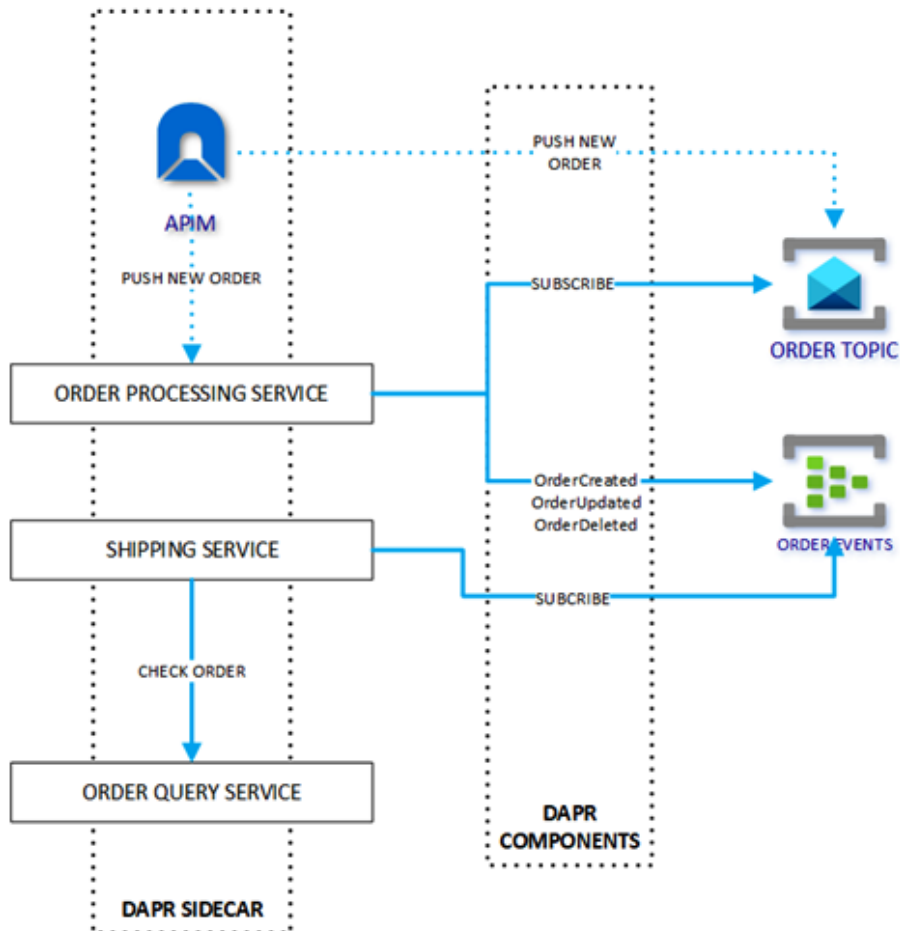


Figure 5.33 – A sample microservices application using Dapr

On the left-hand side of our diagram, we have the following three services:

- The **ORDER PROCESSING SERVICE** receives order requests from two channels. One is an Azure Service Bus topic, and the other one is a direct HTTP/gRPC request to a specific endpoint. Whenever the service creates a new order, it will publish an **OrderCreated** event to Azure Event Hub. All the pub/sub plumbing is ensured by using Dapr components. Direct calls to the order processing service are proxied by the Dapr sidecar.

- The **SHIPPING SERVICE** is a subscriber of the event hub and will be notified whenever an order event lands on Azure Event Hub. It will, in turn, call the **ORDER QUERY SERVICE** to verify that the order still exists and grab extra details about that order. It will then start/modify/cancel the shipping process according to the order event. Direct calls to the query service are proxied by the Dapr sidecar.
- The **QUERY SERVICE** simply returns the searched order out of its fake in-memory order array.

With this small example, we already have quite a lot of interactions with multiple channels: direct invocation and pub/sub.

Let's get started with the actual implementation.

Developing our solution

Now it is time for us to develop our solution! Please review the *Technical requirements* section to perform all of the steps depicted in this section. Let's start with the infrastructure, which is a prerequisite to host our application code.

Deploying the infrastructure

For your convenience, we provided both Bicep and ARM files. They are available at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/tree/master/Chapter05/Code/IaC/>.

The Bicep file is for illustration purposes only. You can download the `chapter05.json` file and deploy it with the following command:

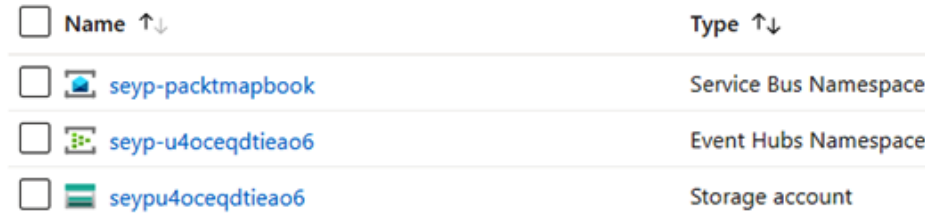
```
az deployment group create --resource-group packt --template-file .\chapter05.json
```

Make sure that you have a resource group named *packt* before running the command, which you can execute with Azure Cloud Shell or Visual Studio Code. Refer to *Chapter 4, Infrastructure Deployment*, for a refresher on this topic, if needed. Upon deployment, you will be prompted to provide a namespace value. Use something concise such as your trigram or a pseudo (one word and no exotic characters). We made sure to concatenate the provided value with a unique string since public PaaS services must use unique names. Alternatively, you can create the resources manually with the Azure Portal. For your information, we require the following services:

- Azure Service Bus
- Azure Event Hub

- Azure Storage, with a container named **packt** within the Blob storage

Ultimately, you should end up with something similar to what's shown in *Figure 5.34*:





<input type="checkbox"/> Name ↑↓	Type ↑↓
<input type="checkbox"/>  seyp-packtmapbook	Service Bus Namespace
<input type="checkbox"/>  seyp-u40ceqdtieao6	Event Hubs Namespace
<input type="checkbox"/>  seypu40ceqdtieao6	Storage account

Figure 5.34 – The resources needed for our scenario

Should you wish to deploy the services manually, make sure that you do the following:

1. Create a blob container, named `packt`, in the storage account.
2. Create a consumer group, named `shipping`, in the event hub.
3. Create a Service Bus topic, named `dapr`, in the Service Bus.

Let's now update our Dapr components with the right connection strings (of our newly created services).

Updating the Dapr components

For the sake of simplicity, the demo ASP.NET Core application can be downloaded or cloned from GitHub at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/tree/master/Chapter05/code>.

Here are the steps to update the components:

1. Locate `OrderService/Components/packtsb.yaml` and replace `<yourconnectionstring>` with the Service Bus connection string, which you can get the following way:
 - a) Connect to the Azure Portal and locate your Service Bus instance.
 - b) View the shared access policies.
 - c) Copy the primary connection string.
 - d) Paste the string into the component file.

2. Locate `OrderService/Components/packteh.yaml` and replace `<yourconnectionstring>` with the Event Hub connection string, which you can get the following way:
 - a) Connect to the Azure Portal and locate your event hub namespace.
 - b) View the shared access policies.
 - c) Copy the primary connection string.
 - d) Paste the string into the component file, and make sure that you leave `;EntityPath=dapreh` untouched.
3. For the `OrderService/Components/packteh.yaml` file, replace `<yoursaname>` with the name of your storage account.
4. Locate `<yoursakey>` and replace it with the primary key of your storage account, which you can get the following way:
 - a) Connect to the Azure Portal and locate your storage account.
 - b) Access the keys.
 - c) Copy the primary key and replace `<yoursakey>` with it.

You can copy both the `OrderService/Components/packteh.yaml` and `OrderService/Components/packtsb.yaml` files to the `ShippingService/Components` folder to replace the files provided by default. Now we are ready to inspect the code, and then we will test the solution.

Getting started with the code

In the following subsections, we will highlight the important parts of the code. You do not need to perform any steps, as they are already part of the provided solution, but we want to provide some explanations.

Adding Dapr to controllers

In the startup class of each service, we have added Dapr to the controllers in the following way:

```
public void ConfigureServices(IServiceCollection services) {  
    services.AddControllers().AddDapr();  
}
```


This injects both the Dapr sidecar, as well as the Dapr client, into our controllers. For any service that uses the pub/sub mechanism, we must ask Dapr to map the subscribers to our topics. This can be done in the `Configure` method as follows:

```
app.UseEndpoints(endpoints =>{
    endpoints.MapSubscribeHandler();
    endpoints.MapControllers();
});
```

For each controller, we can retrieve the Dapr client through the constructor dependency injection, as shown in the following code:

```
private readonly DaprClient _dapr;
public OrderController(ILogger<OrderController> logger,
    DaprClient dapr){
    _dapr = dapr;
}
```

We are good to go with Dapr now. Let's inspect our order processing service.

Inspecting our order processing service

As a reminder, our order processing service is bound to Azure Service Bus and to Azure Event Hub. On top of it, it has a direct method that can be invoked to create orders. Let's now explore the three methods involved in the overall process. Let's start with the Service Bus binding:

```
[Topic("daprsb", "dapr")]
[HttpPost]
[Route("dapr")]
public async Task<IActionResult> ProcessOrder([FromBody]
    Order order){
    _logger.LogInformation($"Order with id {order.Id}
        processed!");
    await PublishOrderEvent(
        order.Id, OrderEvent.EventType.Created);
    return Ok();
}
```

We tell Dapr to bind this method to our component, named `daprsb`, and to subscribe to the `dapr` topic. From there on, whenever a message lands on that Service Bus topic, Dapr will call this method. This is why we also define a route. Upon receipt of a message, we simulate the creation of the order, and we push the `OrderCreated` event to our Event Hub through our `PublishOrderEvent` method, which comes next:

```
async Task<IActionResult> PublishOrderEvent(Guid orderId,
    OrderEvent.EventType type) {
    var ev = new OrderEvent {
        id = orderId,
        name = "OrderEvent",
        type = type
    };
    await _dapr.PublishEventAsync<OrderEvent>(
        "dapreh", "shipping", ev);
    return Ok();
}
```

Notice the use of Dapr's `PublishEventAsync` method to publish our event to our event hub.

Next, we have our HTTP `POST` method, which we want to use as a second channel to create the orders:

```
[HttpPost]
[Route("order")]
public async Task<IActionResult> Order([FromBody] Order
    order) {
    _logger.LogInformation($"Order with id {order.Id}
        created!");
    await _dapr.PublishEventAsync<Order>(
        "daprsb", "dapr", order);
    return Ok();
}
```

Here, we use the `route` attribute, and we get the `Order` object in a parameter. We publish the received order to our Service Bus component. In reality, you should, of course, make a validation. However, for sake of brevity, we will focus only on the communication plumbing. Let's inspect our shipping service.

Inspecting our shipping service

Our shipping service subscribes to the Event Hub to be notified whenever order events land in the hub. If this happens, here is the method that gets triggered:

```
[Topic("dapreh", "shipping")]
[HttpPost]
[Route("dapr")]
public async Task<IActionResult>
    ProcessOrderEvent([FromBody] OrderEvent ev) {
    _logger.LogInformation($"Received new event");
    _logger.LogInformation("{0} {1} {2}", ev.id, ev.name,
        ev.type);
    switch (ev.type) {
        case OrderEvent.EventType.Created:
            if (await GetOrder(ev.id)) {
                _logger.LogInformation($"Starting shipping
                    process for order {ev.id}!");
            } else {
                _logger.LogInformation(
                    $"order {ev.id} not found!");
            }
            break;
        case ...
    }
    return Ok();
}
```

Here again, we use the `Topic` attribute to tell Dapr to use our `dapreh` component and to subscribe to the `shipping` consumer group. Notice how we manually filter the event type (`Created`, `Updated`, or `Deleted`). We do this because neither the Event Hub Dapr component nor Event Hub consumer groups have filtering capabilities. By using the `shipping` consumer group, we, of course, reduce the scope of events.

Next comes our `GetOrder` method, which calls our order query service to double-check the existence of the order. In reality, you would really perform the shipping with the order details:

```
async Task<bool> GetOrder(Guid id) {
    HTTPExtension ext = new HTTPExtension();
```

```
ext.Verb = HTTPVerb.Get;
try{
    await _dapr.InvokeMethodAsync<object, Order>(
        "orderquery",
        id.ToString(), null, ext);
    return true;
}
catch (Exception ex){
    if(((Grpc.Core.RpcException)ex.InnerException)
        .StatusCode == Grpc.Core.StatusCode.NotFound)
        return false;
    //else ==> should handle the other cases or rely on
    // retry policies of a service mesh
}
return false;
}
```

Here, we use Dapr's `InvokeMethodAsync` method to call our `orderquery` service. We check whether or not the order exists before we return `true` or `false`. Note that all the traffic for pub/sub and direct service calls are handled over gRPC just by using Dapr, because Dapr enables both gRPC and HTTP by default. Let's now look at how to test the code.

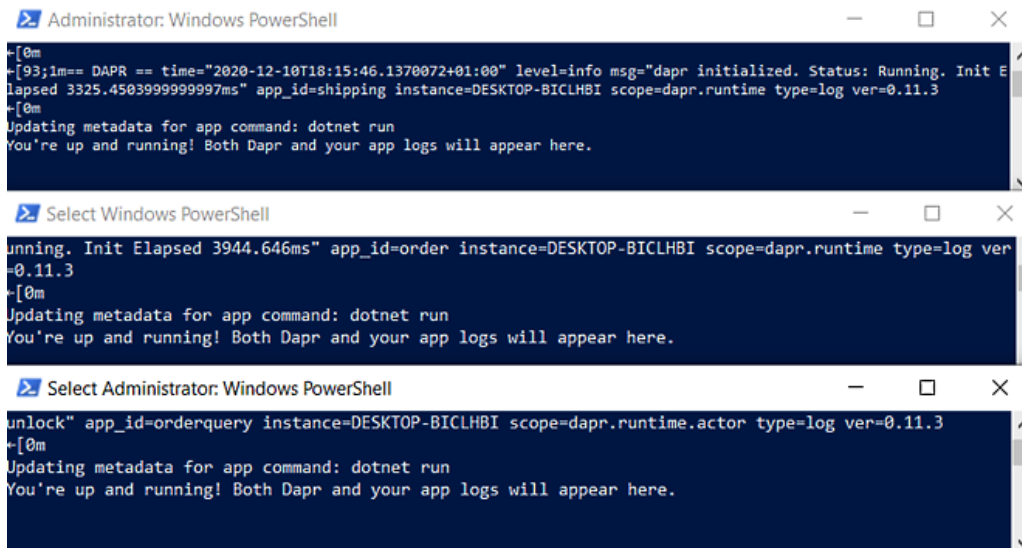
Testing our solution

Finally, it's time to test our code! Here are a few steps that are required to test the solution:

1. Open the `Packt-Microservices-Dapr` solution file with Visual Studio 2019.
2. Build the solution.
3. Open three PowerShell (or DOS) windows. Change the directory of each window to a corresponding service, for example, `C:\Users\steph\OneDrive\Images\maps\packt\chapter5\Code\Packt-Microservices-Dapr\OrderService`.
4. Run the following: `dapr Init`.
5. In the order processing window, run the following: `dapr run --app-id order --components-path ./components --app-port 5002 --port 3503 dotnet run`.

6. In the shipping service window, run the following: `dapr run --app-id shipping --components-path ./components --app-port 5001 --port 3501 dotnet run`.
7. In the order query service, run the following: `dapr run --app-id orderquery --app-port 5000 --port 3500 dotnet run`.
8. Open Fiddler (or Postman), as we are now ready to send a new order request to our order processing service.

If everything goes fine, you should have the three Dapr-enabled services waiting for an activity, as shown in *Figure 5.35*:



```
Administrator: Windows PowerShell
+~[0m
+~[93;1m== DAPR == time="2020-12-10T18:15:46.1370072+01:00" level=info msg="dapr initialized. Status: Running. Init E
lapsed 3325.450399999997ms" app_id=shipping instance=DESKTOP-BICLHBI scope=dapr.runtime type=log ver=0.11.3
+~[0m
Updating metadata for app command: dotnet run
You're up and running! Both Dapr and your app logs will appear here.

Select Windows PowerShell
unning. Init Elapsed 3944.646ms" app_id=order instance=DESKTOP-BICLHBI scope=dapr.runtime type=log ver
=0.11.3
+~[0m
Updating metadata for app command: dotnet run
You're up and running! Both Dapr and your app logs will appear here.

Select Administrator: Windows PowerShell
unlock" app_id=orderquery instance=DESKTOP-BICLHBI scope=dapr.runtime.actor type=log ver=0.11.3
+~[0m
Updating metadata for app command: dotnet run
You're up and running! Both Dapr and your app logs will appear here.
```

Figure 5.35 – Our three services are waiting for an activity

Important note

Note that if you encounter an error, it might be due to ports already being used by other processes on your machine. If this happens, just choose different port numbers than the ones provided. To find out whether a given port has already been used or not, you can use the `netstat -anoc` command.

Now, with Fiddler opened, you can run the following HTTP request (HTTP instead of gRPC), as shown in *Figure 5.36*:

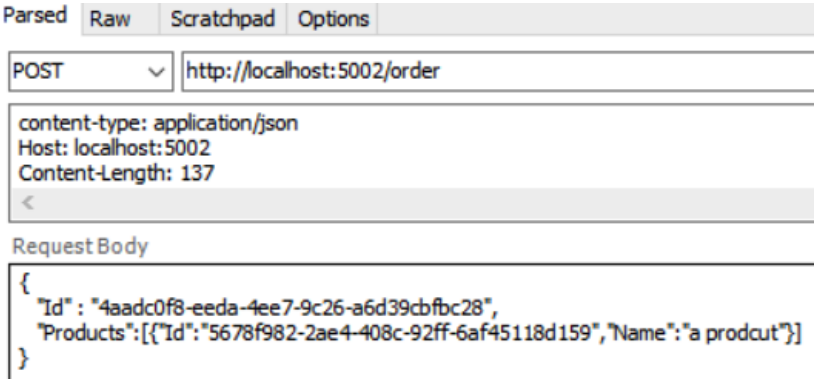


Figure 5.36 – Sending a POST request to our order processing service

The HTTP request causes an activity similar to the one shown in *Figure 5.37* on the service side:

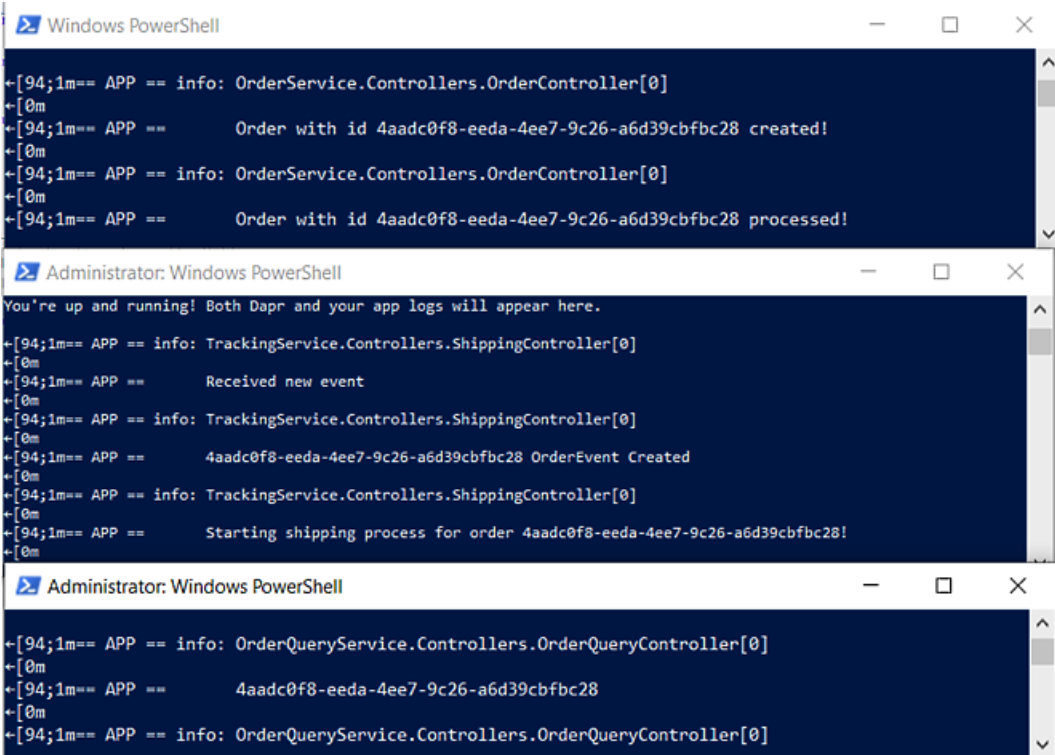


Figure 5.37 – The service activity upon the order creation request

Here, we can see that our order processing service created the order message (pushed to the Service Bus) and then processed it (received from the Service Bus). Once processed, it published an event to the event hub. The event is captured by our shipping service (*Figure 5.37*), which, in turn, calls the order query service. All our services have communicated through the Dapr plumbing. Remember that in our initial diagram (*Figure 5.33*), we had an API gateway that was directly sending messages to Azure Service Bus. In the next section, we will explain how to integrate Dapr with Azure API Management for a zero-code approach.

Combining Dapr and the API gateway of Azure APIM


As we have previously mentioned, API gateways are an integral part of microservices architectures. Now, let's walk you through a very simple zero-code scenario that will leverage Dapr's integration with Azure API Management. Since it involves many different systems, we know for sure that you will probably not have a lab that is ready to really do the exercise. This is why the following is *not* an exercise. We simply want to show you once more what the ecosystem can do for you. Building the environment to support our scenario goes way beyond the scope of the book. However, for your information, here is a list of the systems used in our scenario:

- **A K8s cluster:** This could be AKS, Minikube, or Docker Desktop's K8s.
- **API Management:** This is an instance of Azure API Management with a self-hosted API gateway running on our cluster. More information about self-hosted API gateways is available at <https://docs.microsoft.com/azure/api-management/api-management-howto-provision-self-hosted-gateway>.
- **Dapr:** This is installed into our cluster. More information on how to install Dapr on K8s is available at <https://github.com/dapr/cli> and, more specifically, in the *Install Dapr on Kubernetes* section.
- **Azure Service Bus:** This is our pub/sub service.

Bear with us, even if you have never worked with these technologies. The goal of this example is to make you realize the importance of the ecosystem, and to see how fast you can accomplish more resilient solutions with zero code. Remember that it is your duty as a modern application architect to not let developers reinvent the wheel in their code. In a nutshell, we are going to use the self-hosted API gateway together with some Dapr-specific policies to publish messages to an Azure Service Bus instance.

Our client-facing API will, therefore, not involve any custom code, and it will be more resilient thanks to the extra precautionary measures that we will take.

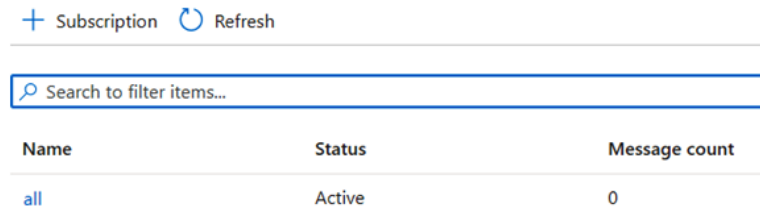
First things first: let's create a new topic for our **packtmapbook** Service Bus instance that we mentioned you in the *Exploring EDAs* section. Please refer to the **dapr** topic in *Figure 5.38*:



Name	Status	Max size
dapr	Active	1024 MB
datasets	Active	1024 MB

Figure 5.38 – Dapr topic

We also created a subscription for this new topic, which subscribes to all the messages (no filter), as shown in *Figure 5.39*:



Name	Status	Message count
all	Active	0

Figure 5.39 – Subscription to all messages from the Dapr topic

We will now create a pub/sub Dapr component with following component file:

```

apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  namespace: packt
  name: daprsb
spec:
  type: pubsub.azure.servicebus
  metadata:
  - name: connectionString
    value: Endpoint=sb://packtmapbook..

```


This Dapr component represents our `packtmapbook` bus. The connection string is incomplete in our example, because we do not want to disclose sensitive information. Note that Dapr components can reference secrets to avoid storing sensitive information directly in the component file, but let's park this aside because it has no influence on what we intent to demonstrate.

We will now add this component to our cluster, as shown in *Figure 5.40*:

```
Select Administrator: Windows PowerShell
PS C:\Users\steph\Source\Repos\azug\dapr\.NET Core\deploy> kubectl apply -f .\packtsb.yaml
component.dapr.io/daprsb created
PS C:\Users\steph\Source\Repos\azug\dapr\.NET Core\deploy> █
```

Figure 5.40 – Adding the Dapr pub/sub component to K8s

We will start our self-hosted gateway and check that its Dapr sidecar has detected our component, as shown in *Figure 5.41*:

```
0
\Users\steph\Source\Repos\azug\dapr\.NET Core\deploy> kubectl logs seyselfhosted-5c488b9b4f-224mq daprd
"2020-12-06T17:50:55.1346578Z" level=info msg="found component daprsb (pubsub.azure.servicebus)" app_id=
```

Figure 5.41 – Dapr detects our component

So far, so good. Our component, `daprsb`, has been found. Therefore, our self-hosted gateway should be able to leverage it. We can create a new API on our Azure API Management instance to verify this assumption (as shown in *Figure 5.42*):

Create a blank API

Basic | Full

- Display name
- Name
- Web service URL
- API URL suffix
- Base URL

Create

Cancel

Figure 5.42 – A new API

Here, we leave the web service URL empty because we will handle the incoming requests with our policies.

Let's create a new operation that will be called by our API clients (see *Figure 5.43*):

The screenshot shows the Dapr UI interface for adding a new operation. At the top, it indicates 'REVISION 1' and 'CREATED Dec 6, 2020, 6:59:28 PM'. Below this are navigation tabs: Design, Settings, Test, Revisions, and Change log. On the left side, there is a search bar for operations, a filter by tags dropdown, and a checkbox for 'Group by tag'. A prominent blue button labeled '+ Add operation' is visible. Below this, the 'All operations' section shows 'No operations to display.' The main area is titled 'dapr > Add operation' and 'Frontend'. It contains several form fields: 'Display name' (order), 'Name' (order), 'URL' (POST method and order path), 'Description' (Post orders), and 'Tags' (e.g. Booking).

Figure 5.43 – A new API operation

Our operation can be targeted through an HTTP POST request against the `dapr-demo/order` path. Now, we need to create the policy that will handle our client requests. Remember that we want to push a message to a Service Bus topic through Dapr. *Figure 5.44* shows the policy that we can apply to our operation:

```
<!-- policies -->
<inbound>
  <base />
  <rate-limit calls="10" renewal-period="60" />
  <publish-to-dapr pubsub-name="daprsb" topic="dapr"
    response-variable-name="dapr-response">@(context.Request.Body.As<string>())</publish-to-dapr>
</inbound>
<backend />
<outbound>
  <base />
</outbound>
<on-error>
  <choose>
    <when condition="@context.Variables.Count > 0">
      <return-response response-variable-name="dapr-response" />
    </when>
    <otherwise />
  </choose>
  <base />
</on-error>
</policies>
```

Figure 5.44 – A policy using Dapr

We start with a **rate-limit** policy to prevent the abuse of our endpoint. We allow a maximum of 10 calls per minute, per subscriber. Afterward, we use the **publish-to-dapr** policy, which targets our **daprsb** component and the **dapr** topic. We capture Dapr's response in the **dapr-response** variable. In the case of errors, we return the **dapr-response** variable if any errors occur. Otherwise, we simply call the default behavior through the **base** keyword. We are now ready to test our setup. With Fiddler's Composer tab, we can try out an HTTP POST request against our endpoint (as shown in *Figure 5.45*):

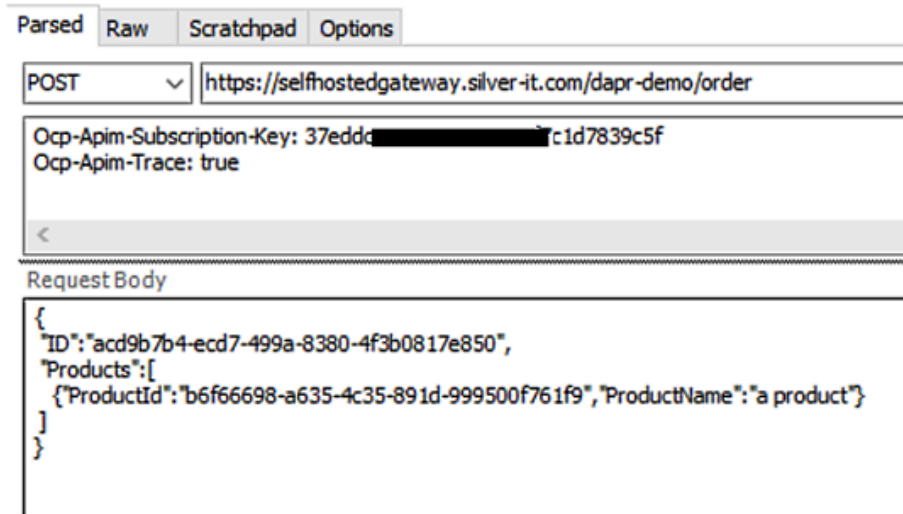


Figure 5.45 – An HTTP POST request against our endpoint

Notice that the subscription key (obfuscated here) passed through the **Ocp-Apim-Subscription-Key** HTTP header. We pass a dummy JSON payload that represents the order in the HTTP POST body. Let's check whether our message landed correctly in our Service Bus topic subscription (see *Figure 5.46*):

Search to filter items...		
Name	Status	Message count
all	Active	1

Figure 5.46 – Message landed in the topic subscription

Sure enough, we see our message, which we can extract through the in-portal Service Bus Explorer. We indeed see our message was received, as shown in *Figure 5.47*:

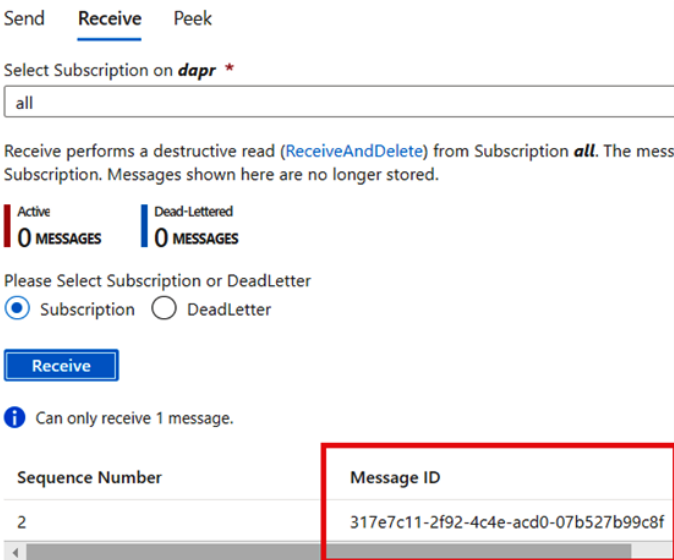


Figure 5.47 – Message inspection with Service Bus Explorer

So, we have the API gateway directly sending messages through Dapr to a Service Bus topic with no code. On top of it, we can take consumption metrics, thanks to the subscription mechanism of Azure API Management. We could add stronger security controls, such as JWT validation or **Mutual Transport Layer Security (mTLS)**. We also have our rate-limiting policy to prevent the abuse of our endpoint, as shown in *Figure 5.48*:

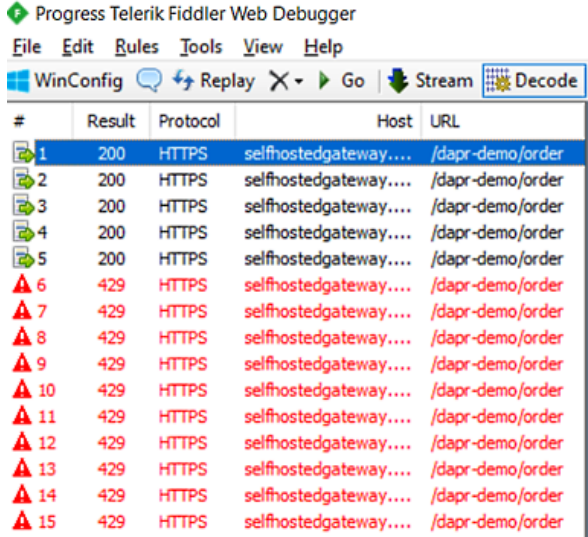


Figure 5.48 – Rate limiting policy in action

We could also add a few validation controls in the policy, such as parsing the incoming JSON body before sending it to Service Bus, and then returning a 400 error in the case of a mismatch. However, we did not do this for the sake of brevity.

Well, what we have achieved is something scalable, secure (we could also add more policies), and traceable, without a single line of code. It was all done only by leveraging the ecosystem. We have offloaded everything to off-the-shelf services. This is one more piece of evidence that the ecosystem prevails in cloud and cloud-native solutions. Because Dapr will definitely be part of tomorrow's ecosystem, we strongly encourage you to also look at the following:

- Dapr with Azure Functions: <https://github.com/dapr/azure-functions-extension>
- Dapr with Logic Apps: <https://github.com/dapr/workflows>

Now, let's recap the chapter!

Summary

In this chapter, we reviewed how ecosystems prevail in modern applications. First, we took a detour to explain cloud and cloud-native development. We then explored our Azure Application Architecture Map. We zoomed in on the data scenarios and the cloud design patterns. Next, we explored EDAs and messaging architectures. Finally, we showed you what it looks like to develop a cloud application with microservices (and you might have just learned a lot about Dapr).

The time of developers crafting everything in code is over. Understanding and relying on ecosystems is the best path to build resilient and scalable solutions in a timely fashion. Professional cloud and cloud-native application architects must step back from the code and look at the broader picture. The application code should only be considered as a placeholder for business logic.

The non-functional requirements should never be handled in code anymore. The Azure and AKS ecosystems must be leveraged to their maximum extent, to maximize the return of investment in those platforms. With container-based applications and some neutral frameworks, such as Dapr, application architects can also develop portable solutions.

In the next chapter, *Chapter 6, Data Architecture*, we will go one step further, toward functional and non-functional delegation to the cloud provider.

6

Data Architecture

In this chapter, we explore how data is processed and stored. We will look at specialized data stores that are uniquely tailored to each dataset and purpose. We will explore traditional **Relational Database Management System (RDBMS)** workloads, as well as modern big data solutions.

We will more specifically cover the following topics:

- Looking at the data architecture map
- Analyzing traditional data practices
- Delving into non-traditional data services
- Diving into big data and AI services
- Getting our hands dirty with a near real-time data streaming use case

This chapter will provide you with a detailed understanding of different data services and a good overview of big data and **Artificial Intelligence (AI)**. You will also gain some hands-on experience of how to process data in real time with Azure.

Let's now review the technical requirements.

Technical requirements

If you want to practice the explanations provided in this chapter, you will need the following:

- **An Azure subscription:** To create your free Azure account, follow the steps explained at <https://azure.microsoft.com/free/>.
- **A Power BI workspace (with a Power BI Pro license):** To create a free Power BI account, follow the steps explained at https://app.powerbi.com/signupredirect?pbj_source=web.
- **A shell system:** This could be DOS, PowerShell, or even Bash on Linux. This is required to run the executable .NET Core app that we have built for you toward the end of this chapter. We will provide explanations on how to start the application with DOS.

All the code and maps (in full size) used in this chapter are available at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/tree/master/Chapter06>.

Let's start with our data architecture map.

The CiA videos for this book can be viewed at: <http://bit.ly/3pp9vIH>

Looking at the data architecture map

It is no secret that every system and application deals with data, and it is no secret that the importance of data is growing year after year, especially with the rise of AI. Data volumes are higher than ever before. Companies need to find ways to store data efficiently and at a reasonable cost while being able to get insights from it. In this chapter, we will browse the vast data landscape of Azure. A reduced version of our data architecture map is shown in *Figure 6.1*:

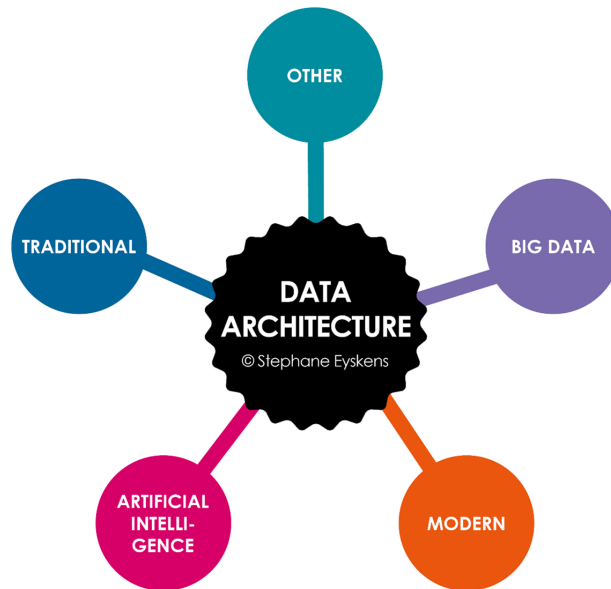


Figure 6.1 – The data architecture map (reduced)

Important note

To see the full data architecture map (Figure 6.1), you can download the PDF file at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/blob/master/Chapter06/maps/Data%20Architecture.pdf>.

Our map has five top-level groups:

- **BIG DATA:** The public cloud is probably the only viable option that deals with real big data. However, big data services can also be used to our advantage with smaller amounts of data. We will see an illustration of this in our *Getting our hands dirty with a near real-time data streaming use case* section.
- **MODERN** and **TRADITIONAL:** We wanted to split modern and traditional data concerns, because Azure allows you to use both. For example, we traditionally refer to **ETL (Extract Transform Load)**, but its modern counterpart is **ELT (Extract Load Transform)**. We traditionally work with relational database engines, while their modern counterparts often rely on NoSQL.
- **OTHER:** This top-level node regroups data-related cross-cutting concerns.
- **ARTIFICIAL INTELLIGENCE (AI):** AI certainly represents the modern way of extracting insights out of data.

As always, this map is not the holy grail, but it will surely help you find your way in the vast Azure data landscape. Let's start with the traditional data practices in Azure.

Analyzing traditional data practices

In this section, we will review services that belong to the traditional data world. Our purpose is to reassure you that what we can do on-premises can also be done the same way in the cloud. Moving data workloads to the cloud does not necessarily mean that you have to completely reinvent yourself and your way of working.

Let's first clarify that we do not use the term *traditional* in a pejorative (nor negative) way. In many situations and for many enterprises, using traditional techniques still provides full satisfaction, and you should not necessarily move to more modern technologies just for the sake of it. (Our point has been made!) *Figure 6.2* displays our zoom-in on traditional data practices:

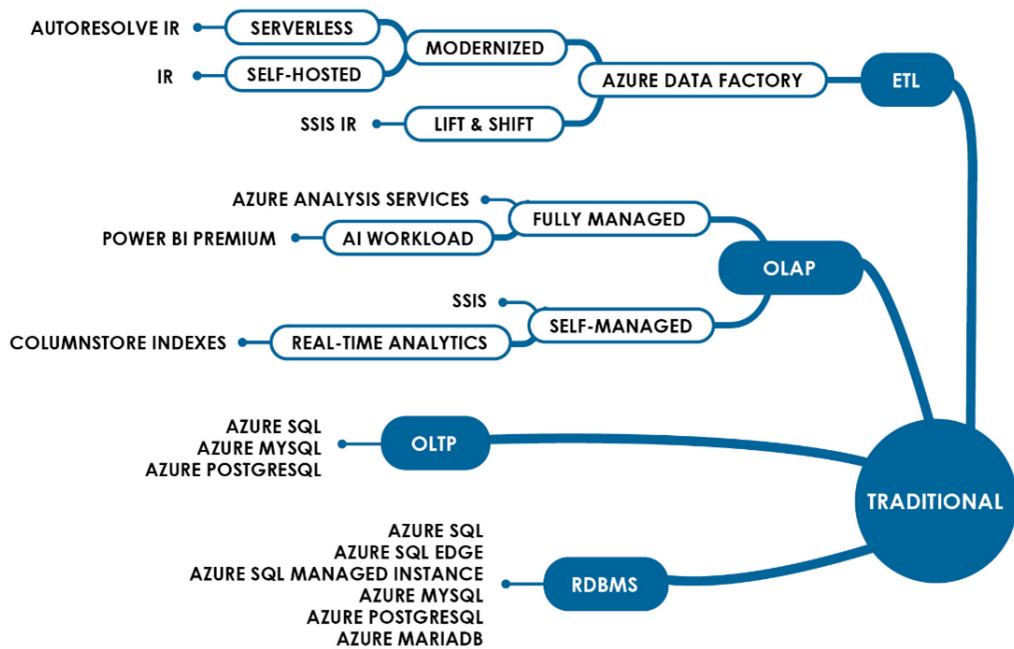


Figure 6.2 – Traditional data practices in Azure

The **TRADITIONAL** node regroups all traditional practices that we've typically used for decades, for which Azure also has a bunch of services that we will discuss in the following subsections. Let's start with **Online Analytical Processing (OLAP)** and **Online Transactional Processing (OLTP)**.

Introducing the OLAP and OLTP practices

OLAP and OLTP are typical **Business Intelligence (BI)** practices. We have been using the entire SQL Server BI stack for years. This stack is typically composed of **SQL Server Reporting Services (SSRS)**, **SQL Server Integration Services (SSIS)**, and **SQL Server Analysis Services (SSAS)**. *Figure 6.3* shows how you can continue to use similar technologies in Azure:

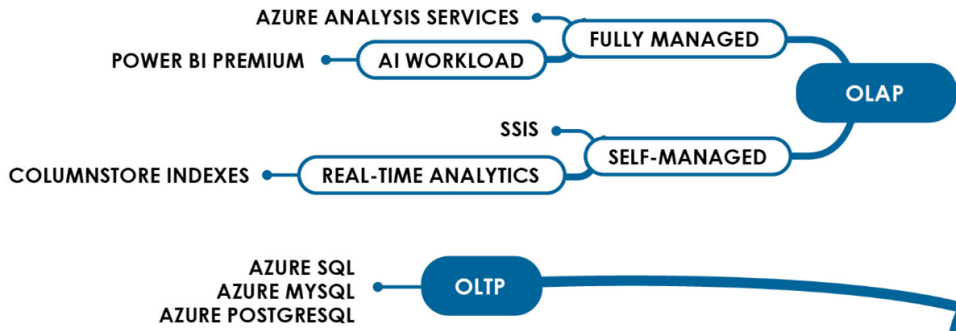


Figure 6.3 – OLAP and OLTP

Under the OLAP node, we see that SSAS is still available as a self-managed service, and that **AZURE ANALYSIS SERVICES** is its fully managed equivalent. One of the benefits of going to the cloud is the ability to delegate infrastructure and scalability concerns to the cloud provider. You should favor **AZURE ANALYSIS SERVICES** Analysis Services or **Power BI Premium** (<https://docs.microsoft.com/power-bi/admin/service-premium-what-is>) whenever possible, so as to speed up time to market and reduce your operational burden. Power BI Premium comes with extra AI features compared to Power BI Pro. Should you not be happy with managed offerings, you can still rely on self-managed SSIS and SQL Server column store indexes. Let's now explore the ETL capabilities in Azure.

Introducing the ETL practice

Extract Transform Load (ETL), illustrated in *Figure 6.4*, is a traditional practice that consists of extracting data from a source, validating and transforming it on the fly, and then loading it into a destination:

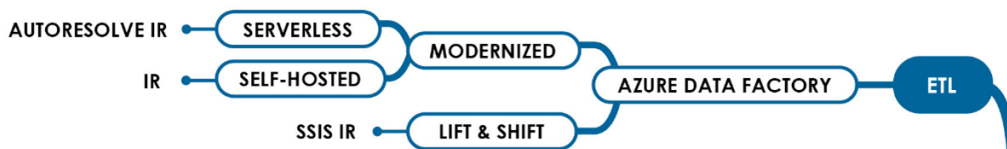


Figure 6.4 – ETL in Azure

Figure 6.4 shows that Azure offers **Azure Data Factory (ADF)** to perform such tasks. ADF helps build visual pipelines with the ADF authoring portal. Every ADF pipeline is associated with an integration runtime. The serverless offering makes use of the **AutoResolveIntegrationRuntime**, which is hosted by Microsoft inside a network perimeter that is not under your control. This IR flavor is the most cost-effective and fully managed solution, but it is often discarded because the source and destination systems cannot be firewalled with company-owned IP ranges/perimeters. A more recent possibility that is still in preview as of 01/2021 consists of using the auto-resolve runtime with the managed virtual network option. This allows you to connect to **Platform-as-a-Service (PaaS)** services sitting behind private endpoints.

For that reason, Azure also ships with the **Self-Hosted Integration Runtime**, which is functionally equivalent, but as the name indicates, it is self-managed. At the time of writing, this runtime is available in the form of a Windows service that must be installed on one or more virtual machines. This, of course, kills cost-efficiency and makes you entirely responsible for high availability and disaster recovery. That is the cost of a firewall rule! Similarly, you can use the **SSIS Integration Runtime** for lift-and-shift scenarios of your on-premises SSIS to the cloud. Let's now explore RDBMSes.

Introducing the RDBMS practice

In *Chapter 5, Application Architecture*, we stressed the importance of choosing an appropriate data store, and we recalled the difference between ACID and BASE. Although RDBMSes are traditional, they will probably remain in use forever, because most enterprise businesses require ACID capabilities. Azure ships with many RDBMSes as shown in *Figure 6.5*:



Figure 6.5 – RDBMSes in Azure

The native Microsoft offering consists of Azure SQL Database, Azure SQL Edge, and Azure SQL Managed Instance. Azure SQL Database is part of the public PaaS offering, while Azure SQL Managed Instance is a fully dedicated SQL server instance, managed by Microsoft. The main historical reason for choosing this option was to avoid any public exposure. Now that Azure SQL public endpoints can be fully removed, thanks to Private Link (more on this in *Chapter 7, Security Architecture*), there are fewer reasons to go for a fully dedicated instance.

Azure SQL Edge, as its name indicates, is used for all sorts of devices that operate at the edge, meaning the entry point of an enterprise network. The purpose of the edge is to react faster to local changes and diminish back and forth roundtrips between the cloud and the corporate network. Edge technologies are also usually resilient to a loss of connectivity. SQL Edge is a data service that can be installed on such devices. The other RDBMS services (such as Azure Database for MySQL, Azure PostgreSQL Databases, and Azure Database for MariaDB) are also fully managed by Microsoft. Now that we've covered a bunch of traditional systems and practices, let's delve into modern data services and practices.

Delving into modern data services and practices

As stated in the previous section, you should not rush to newer technologies for the sake of it. However, the good news is that most of these technologies are available at an affordable price, so you should at least explore them. Although we try to set services under a certain category, some of them span multiple categories. So, you shouldn't consider their positions on the map to be official or the only way to organize them. For example, Azure Storage spans at least the modern and big data categories. *Figure 6.6* shows Azure's vast modern data landscape, which we will review in the upcoming subsections:

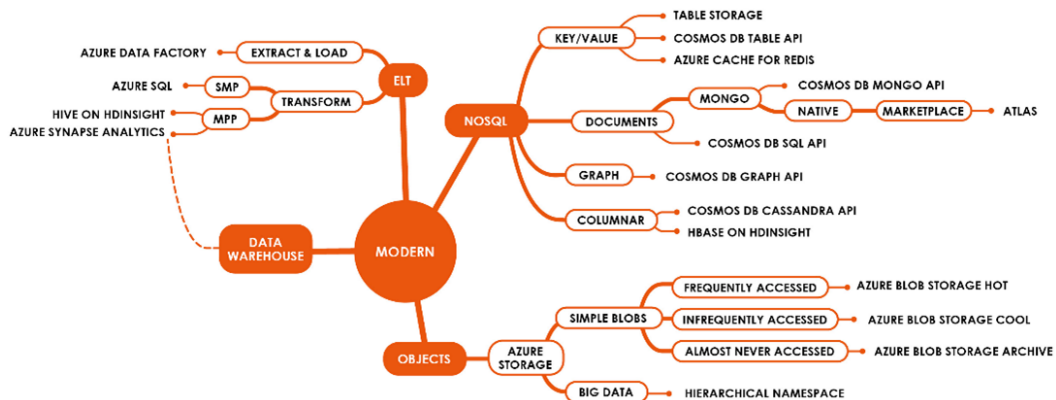


Figure 6.6 – The modern data landscape in Azure

Let's now explore the **ELT** category, which is the modern counterpart of ETL.

Introducing the ELT practice

ELT is quite similar to ETL, with one notable exception: the transformation step. As we saw earlier, in ETL, every step is entirely handled by the ETL pipeline. In ELT, the pipeline is mostly used to copy data from one source to a destination. The transformation aspect is handled by the destination itself, which requires a data store that can scale accordingly and has native transformation capabilities. *Figure 6.7* shows ELT in Azure:

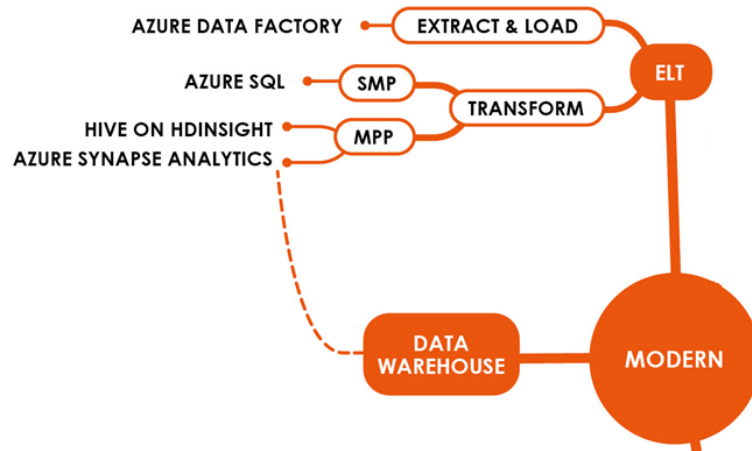


Figure 6.7 – ELT in Azure

To ensure the *extract* and *load* parts of ELT, you can still rely on Azure Data Factory. For the transformation part, you may use Azure SQL Database for **SMP (Symmetric Multiprocessing)**, and Hive on HDInsight or Azure Synapse Analytics for **MPP (Massively Parallel Processing)**. The principle of SMP is to share a single system bus across multiple processors, and let each processor handle a dataset independently from the others, while the computer resources are shared with other processors equally.

The principle of MPP is to split datasets across computers, so as to let each computer run its own dedicated resources (which makes it a much more scalable approach). To be clear, Azure SQL is able to do both SMP and MPP, but **Azure Synapse Analytics** (formerly known as **Azure SQL Data Warehouse**) can handle that at a higher scale. Therefore, Synapse Analytics is the tool of choice for MPP workloads.

Let's now explore the NoSQL services.

Exploring NoSQL services

NoSQL is becoming more and more present in the enterprise. As we saw in the previous chapter, NoSQL fits many purposes and is well suited for distributed and modern applications. *Figure 6.8* shows the different NoSQL options:

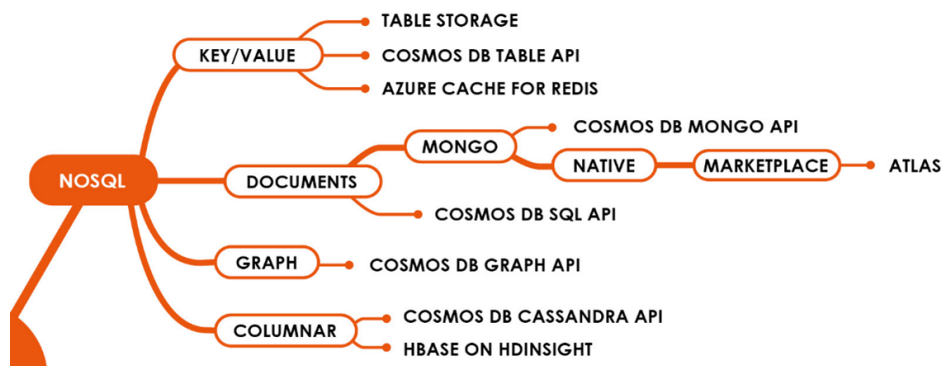


Figure 6.8 – NoSQL services in Azure

We see different areas such as key/value stores, document stores, graph stores, and columnar stores, which we will review in the coming subsections.

Let's start with the **KEY/VALUE** top-level group in the preceding diagram.

Learning about key/value stores

Table Storage, **Cosmos DB Table API**, and **Azure Cache for Redis** can all be used as **key/value** stores. All of these stores are based on strong consistency. They can be used for apps that require high throughput and massive interactions with the data stores. Table Storage is Azure Storage's historical table store, and it is one of the first Azure services in Azure's history.

Cosmos DB's Table API is the premium flavor of Table Storage. It comes with dedicated throughput and single-digit millisecond latencies, but of course, it also comes at a higher price. Cosmos DB Table API should therefore mostly be considered for mission-critical workloads that have specific availability and latency requirements.

Azure Cache for Redis is mostly used to implement the cache-aside pattern, which we covered in the previous chapter. Let's now look at the various document stores.

Learning about document stores

The only pure Azure native document store is **Azure Cosmos DB**. Cosmos DB is Azure's NoSQL Swiss Army knife that you can use for almost everything. The preferred option when working with Cosmos DB is to use the **SQL API**. We recommend it because it's the fastest and lets you write the most advanced queries.

Should you need to lift and shift an existing **MongoDB** database, you might leverage the **MongoDB API** of Cosmos DB, but you should pay attention to the required MongoDB API. Indeed, the features of Cosmos DB are often lagging behind the native MongoDB offering. At the time of writing, Cosmos supports MongoDB API 3.6, while the native MongoDB is at version 4. If you really prefer to work with a native MongoDB offering, you can leverage **Atlas**, a marketplace solution for which there is a free offering available called the **M0 tier**.

Let's now explore the remaining NoSQL store types.

Looking at other store types

For both graph and columnar **stores**, you can still rely respectively on the **Cosmos DB Graph API (Gremlin)** and **Cosmo DB Cassandra API**. As an alternative, you can run **HBase on HDInsight**. Besides NoSQL stores, we also rely on object stores, so let's have a look at them.

Learning about object stores

Azure Storage, and more particularly **Azure Blob Storage**, is the de-facto choice for storing blobs. The type of blob storage differs according to the usage required. See *Figure 6.9* for the object stores in Azure:

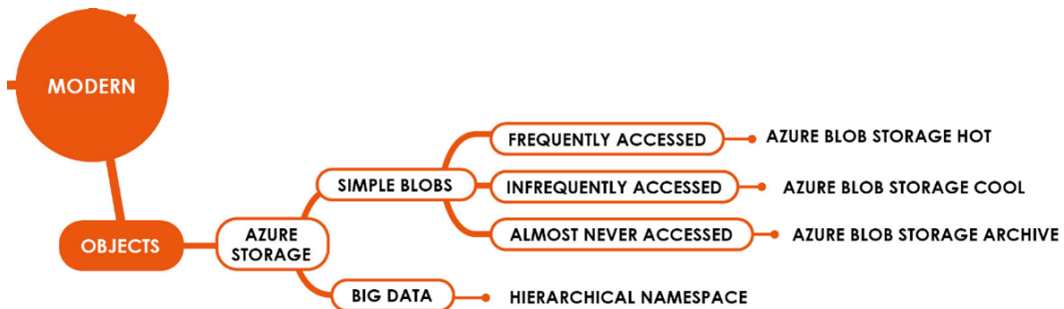


Figure 6.9 – Object stores in Azure

For blob-only storage, you can choose the Hot, Cool, or Archive pricing tiers. They all correspond to a certain usage frequency and come at different prices. To work with **data lakes**, as a prerequisite, you should enable the **Hierarchical Namespace** feature of Azure Blob Storage. This step converts a normal blob storage into a data lake.

Azure data lakes are built on top of Azure Blob Storage, but they are not enabled by default. You might still see some old references to **Azure Data Lake Storage Gen1**, which were the first generation of data lakes in Azure. Every greenfield project should leverage **Azure Data Lake Storage Gen2**, which is enabled by the hierarchical namespace feature described earlier. By the way, this is the reason why we placed it here and not under the big data category, although data lakes are used for big data. Let's now also explore our big data top-level group, which is also a modern way of dealing with data, but it deserves its own dedicated top-level group.

Diving into big data services

Big data goes way beyond traditional data technologies, so it could have been placed under the *MODERN* top-level group. However, as stated before, it is easier to make it a separate group for the sake of clarity. In a nutshell, big data deals with data sets that are too large to be handled by traditional data technologies. The good news, however, is that the opposite is possible. You can use big data technologies to handle smaller volumes, and it is doable at an affordable price in the cloud. *Figure 6.10* shows the big data landscape in Azure:

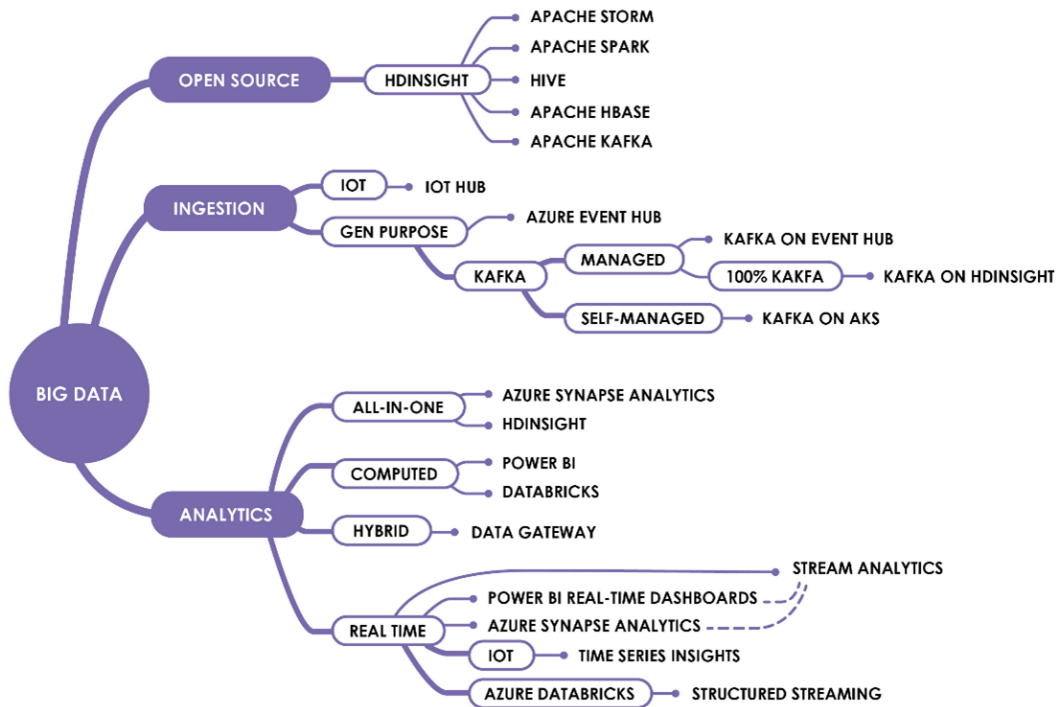


Figure 6.10 – Big data in Azure

As a preamble, we can say that Azure's data service masterpiece is **Azure Synapse Analytics (ASA)**. ASA can be used for whatever data purpose you may have with Azure. Therefore, it's impossible to position it in a very specific place on the map. So, if you know nothing about Azure data services, you should consider ASA an all-in-one data service.

We have divided the big data category into three sub-groups: **OPEN SOURCE**, **INGESTION**, and **ANALYTICS**. Let's explore them now. We'll start with the **INGESTION** group.

Ingesting big data

Ingesting big data means that you're potentially ingesting millions of messages per second, but not all services have that capability. *Figure 6.11* shows the services you can use in Azure to sustain such high volumes:

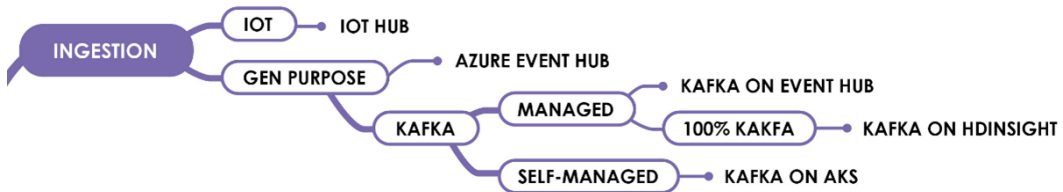


Figure 6.11 – Big data ingestion in Azure

For **Internet of Things (IoT)** scenarios, **Azure IoT Hub** is the most complete offering. It supports both **Device to Cloud (D2C)** and **Cloud to Device (C2D)** style of communications, and none of its alternatives do. You also have more advanced device-related information thanks to the **Device Twins**. Note that you can also use both **Azure Event Hubs** and **Kafka on HDInsight**, but they have no IoT-specific features. IoT Hub also supports the IoT protocol gateway and IoT Edge module. It is a full set of services dedicated to IoT solutions.

For general purpose ingestion, you may rely on the native Azure Event Hubs service, which can ingest millions of events and messages per second. Azure Event Hubs may have one or more consumer groups, which can handle messages as they come. It is also often used in conjunction with **Azure Stream Analytics** to perform on-the-fly analytics and to route data accordingly.

If you need to work with the Kafka protocol, you may opt for **Kafka on HDInsight**, which is fully managed by Microsoft, or **Kafka on Event Hubs**. Kafka on HDInsight supports 100% of the Kafka features. Choosing between Kafka on HDInsight and Kafka on Event Hubs should be determined based on the features you need and the associated costs. Kafka on HDInsight is much more expensive. You may as well use a self-managed version of Kafka on AKS, or even directly on virtual machines. Doing so might be cheaper (although that's not guaranteed), but you're also then responsible for the high availability and disaster recovery duties. Of course, satellite services, such as Azure Data Factory and Azure Storage, may play a role in the ingestion mechanics.

Let's now explore some big data analytics services.

Exploring big data analytics

The purpose of data analytics is to extract useful insights from data, and ultimately help the decision-making process. As stated in the preamble, **ASA** is an all-in-one data service. It integrates with many other services and is the former Azure Data Warehouse solution. More than just a rebranding, ASA aims to regroup most traditional/modern and big data concerns into a single service. On top of its ELT capabilities, ASA lets you analyze data at scale, leverage both machine learning and deep neural networks, and perform near real-time data analysis. The same goes for HDInsight, but with its open source roots. *Figure 6.12* shows the big data analytics services in Azure:

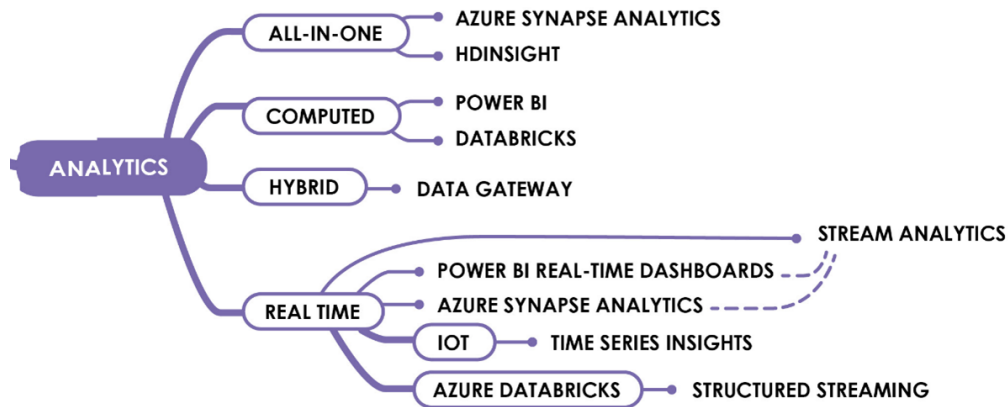


Figure 6.12 – Analytics on Azure

We will talk about HDInsight in the *Azure-integrated open source big data solutions* section. ASA is often compared to **Azure Databricks**, yet another analytics tool. In a nutshell, we can say that Azure Databricks really focuses on an improved version of Spark and is intended to be used by data scientists only. Unlike ASA, Azure Databricks is also not suitable for data warehouses. They both integrate with data lakes. However, data lake integration with Azure Databricks is optional. Finally, **Structured Streaming** is available in Databricks to perform near real-time analysis.

Power BI can be used for reporting purposes; it has real-time APIs that can surface data onto real-time dashboard tiles. Power BI real-time datasets are often directly provisioned by Stream Analytics jobs, or they can be fed using Power BI's real-time APIs. Overall, Power BI exists to meet your data visualization needs.

When it comes to pure analytics, ASA connects inputs and outputs. ASA can apply on-the-fly data queries and take real-time decisions based on the query outcomes. For IoT analytics, **Time Series Insights** (now at Gen 2), is one of the best choices. The reason why Time Series Insights is a good fit for IoT is because IoT devices typically send telemetry data at regular time intervals. Depending on the time interval and the number of devices, you may quickly be overwhelmed by the amount of data. Time Series Insights can help you detect anomalies and hidden trends in your data. It also ships with an SDK that makes it easier to surface data into visual charts.

An **Azure data gateway** is necessary when you want to connect Power BI (as well as the entire Power Platform and Azure Logic Apps) to on-premises data sources, even if you already have a hybrid Azure setup (which we discussed in *Chapter 3, Infrastructure Design*). If you have ExpressRoute, you can configure the data gateway to bypass the internet. If you don't have ExpressRoute, the data gateway will typically reach out to Azure over an internet connection. The gateway itself can be installed on any on-premises machine (or even in a DMZ), provided this machine has access to the targeted on-premises data store. At the time of writing, the gateway can only be installed on Windows Server and does not currently ship as a container. And lastly, like many other Azure agents, the gateway only requires outbound connectivity, not inbound, which significantly simplifies the aspects of firewall configuration.

Let's now look at HDInsight.

Azure-integrated open source big data solutions

Azure HDInsight allows you to run the most important Apache data services shown in *Figure 6.13*. Since our book is about Azure and related Microsoft technologies, we will not delve into the Apache stack, but we wanted to inform you about this important possibility. In 2020, the native open source Apache building blocks transitioned to the **Microsoft Distribution of Hadoop (MDH)**. For the open source purists out there, this might be bad news. However, for the rest of us, this simply means you get better support from Microsoft and better integration with Azure:

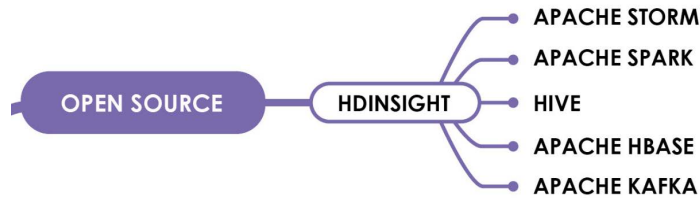


Figure 6.13 – HDInsight open source support in Azure

A concrete example that is born out of this transition to MDH is **SparkCruise**, which boosts Apache Spark queries by building materialized views automatically. This way, it can let subsequent queries consume from the materialized views, instead of computing again the results. HDInsight is also an all-in-one service, and there is probably no obvious reason to opt for HDInsight or Synapse, other than the fact that Synapse is likely going to be better integrated with the rest of the Azure ecosystem.

We will now look at some AI data solutions.

Introducing AI solutions

AI has been on everyone's lips for many years now. In this section, we will review the most important AI concepts and their corresponding Azure services.

Figure 6.14 is a summary of Azure's AI landscape:

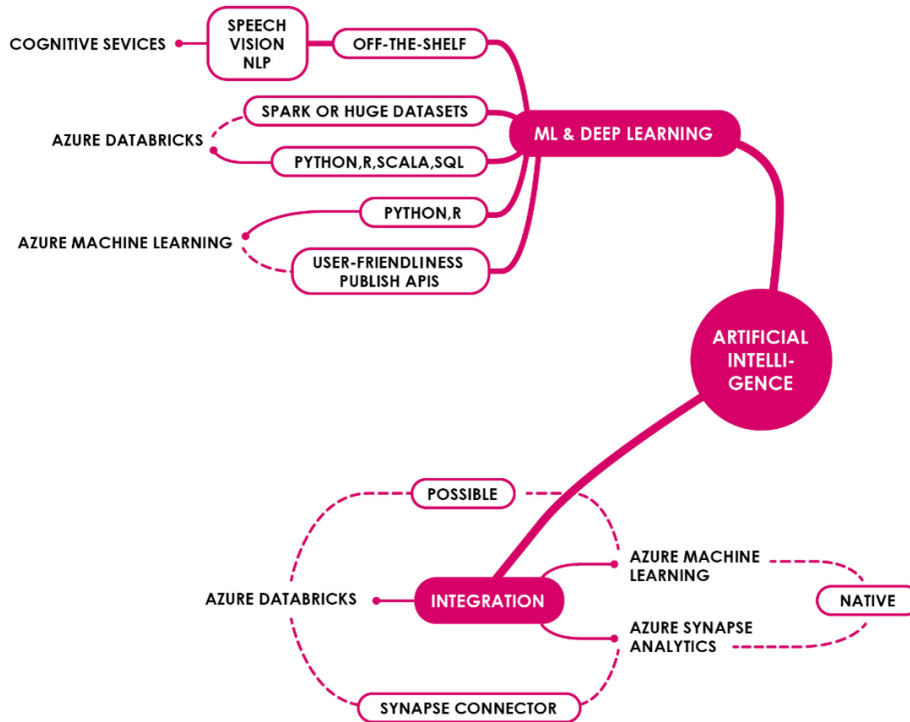


Figure 6.14 – AI solutions in Azure

We say this is a summary because Azure Cognitive Services (alone) can be regrouped into about 20 different services. We already talked about most of these services, but this time we will take the AI-specific angle. Let's start with the machine learning and deep learning options.

Understanding machine learning and deep learning

The services depicted in Figure 6.15 are quite close, in terms of capabilities, and they are all intermingled. This makes it very hard to position them for a specific use case:

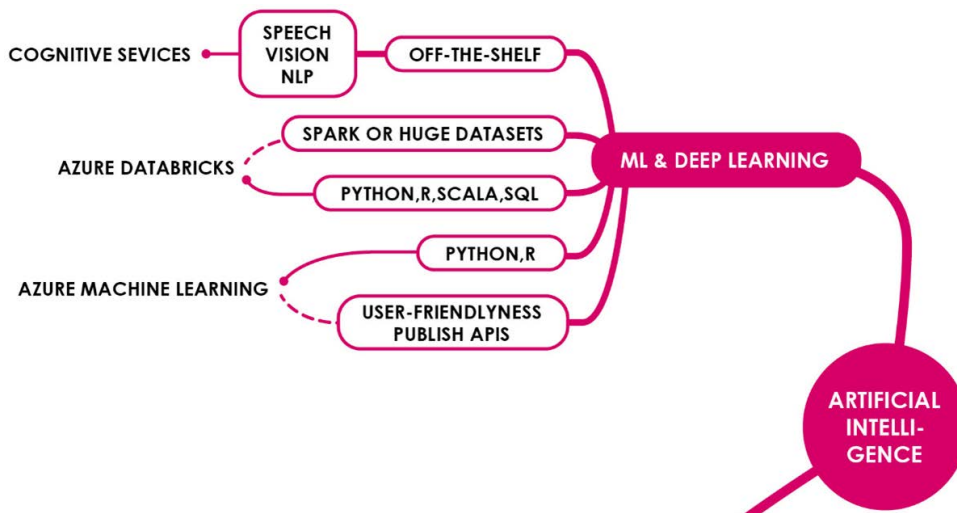


Figure 6.15 – Machine learning in Azure

The only no-brainer is **Azure Cognitive Services (ACS)**, a full set of pre-built AI capabilities, which are made available to the average developer through REST APIs. The biggest benefit of ACS is that you do not need any specific AI skills to leverage it. The underlying models are ready to be deployed as a new instance for you to use. They respond to mainstream AI needs, such as **Optical Character Recognition (OCR)**, speech recognition, and **Natural Language Processing (NLP)** requirements, such as text translations, entity recognition, and so on.

ACS makes it easy to empower your applications with AI features. **Azure Cognitive Search** embeds AI capabilities into its search engine, thanks to its cognitive peers, at no extra cost. Another strength of ACS is that you can also easily get an instance tailor-made for your business. For example, you can spin up an instance of **LUIS (Language Understanding)**, and then train it with your own business jargon and patterns in a matter of hours. LUIS will quickly recognize your custom business entities and can even be trained and used by business users.

The same applies to the **Custom Vision** service, which lets you build custom image classifiers in minutes. The cherry on the cake is that most ACS services can also be hosted within containers, which makes it easy to deploy them on edge devices or in any containerized environment.

Things get more complicated when we want to compare **Azure Databricks** and **Azure Machine Learning (AML)**, because they are both very good products. Azure Databricks supports more languages than AML and is Spark-based. Azure Databricks is particularly well suited for very large datasets. Unlike ACS, Azure Databricks is intended to be used by data scientists only, because it is mainly built on Python, R, and Scala, which ship with machine learning libraries. This makes Databricks harder to use for the average developer who is usually unacquainted with these specific languages and libraries.

As you can imagine, it is a lot easier to get started with AML thanks to its Azure Machine Learning Studio user interface, which lets you easily build machine learning workflows. It also does a very good job of comparing different models and showing their corresponding scores. Another strength of AML is the fact that it lets you publish APIs of models that were trained with another engine. For example, you might publish an R model that was generated on-premises, and then make it available as a web service that you could proxy (or not) with Azure API Management.

Overall, it is hard to choose between AML and Azure Databricks. External factors, such as costs, skills, and company culture, may become the deciding factors. The good news is that you do not necessarily have to choose between them, as we will see in our next section.

Integrating AI solutions

Figure 6.16 shows the different interactions between AML, Azure Databricks, and ASA:

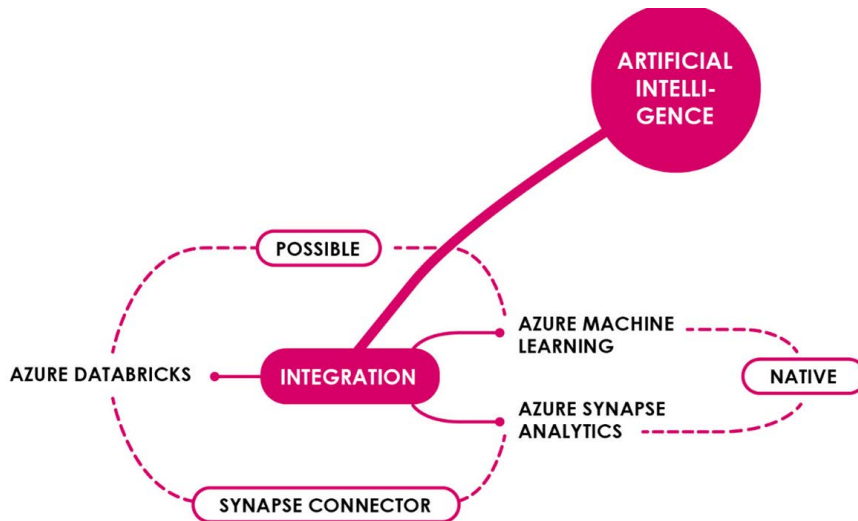


Figure 6.16 – AI analytics integration

As stated in the previous section, these services are not mutually exclusive. For example, you can leverage the **Synapse Connector** to use ASA as a data source for Azure Databricks. ASA has native integration with AML. It lets you consume models that are trained in AML, but that are accessed by ASA (through the **Automated ML** feature). Integration between the two systems is secured by Azure Active Directory. Lastly, even Azure Databricks and AML can integrate. To get the best of both worlds, you might decide to prepare and train your data in Azure Databricks, but then serve it in AML.

So far, we mostly covered data technologies. Whether we deal with traditional or modern technologies, there are some transversal concerns that we will cover in our next section.

Dealing with other data concerns

In this section, we will review some cross-cutting data concerns, no matter whether you make use of traditional or modern data techniques. *Figure 6.17* illustrates some transversal needs, whether you are in a traditional, modern, or big data world:

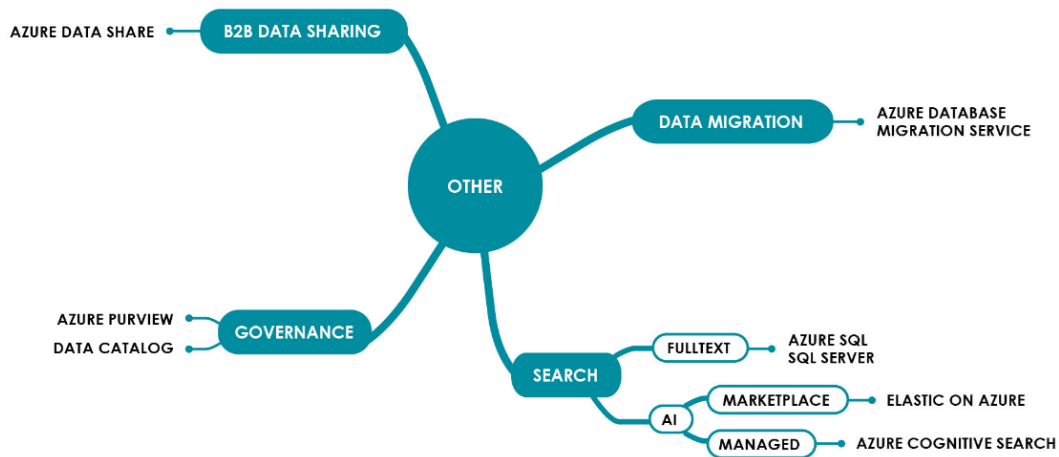


Figure 6.17 – Cross-cutting data concerns

Let's start with search.

Introducing Azure Cognitive Search

Azure ships the **Azure Cognitive Search** service. It used to simply be named **Azure Search**, but it got rebranded to the new name because it also now encompasses other AI capabilities brought by the different cognitive services. Azure Cognitive Search is very well integrated with the other Azure services, which means that you can plug any Azure data store (Cosmos DB, Table Storage, Azure SQL, and so on) into it.

The cognitive part enriches the search contents. For example, the NLP engine will detect entities and extract key phrases and sentiment, while the OCR functionality will extract text from indexed images. AI enrichment is particularly useful with unstructured content. You can use built-in indexers, such as the Azure SQL indexer, which will feed Azure Cognitive Search with SQL data sources. You can also use dynamic indexers, which you can create with various applications and feed with data as it comes. The search engine exposes OData capabilities, and two **Lucene**-based query languages, which are **Simple Query Parser** and **Lucene Query Parser**.

Elastic on Azure is a marketplace alternative, which is also Lucene-based and has open source roots. Feature-wise, they are both comparable, although Elastic is a bit richer. For example, it supports *many* more data types than Azure Cognitive Search. It also includes the concept of watchers, which have no equivalent in the Azure Cognitive Search world. But once again, you should first look at your requirements and then choose the most appropriate solution.

Back in *Figure 6.16*, we split AI and full-text search, but of course, both Azure Cognitive Search and Elastic also perform full-text indexing. We simply wanted to indicate that for mere full-text indexing, you can also rely on the built-in Azure SQL full-text search. Sometimes, sharing data is necessary, and that is what we will discuss in our next section.

Sharing data with partners and customers (B2B)

B2B data sharing can be achieved using **Azure Data Share**. With Azure Data Share, the data provider can send invitations to a data consumer in order to let the consumer directly access the source store, or to send a snapshot of the shared data to the consumer side. The consumer can choose the target receiving the store, which is independent of the source data store type. You can share Azure Blob Storage data and let the consumer decide to receive that data into a data-lake-enabled target. If you opt for in-place access, the consumer receives a link to access your data with read-only permissions, thus preventing the creation of snapshots, which are not free.

Sharing data is interesting, but what about migrating your data? We will explore data migration in our next section.

Migrating data

Azure Database Migration comes in handy for on-premises-to-cloud or cloud-to-cloud data migrations. The migration process counts three phases: discovery, migration, and post-migration actions. The complexity of the discovery process depends on the type of migration you make, either homogenous or heterogenous.

Homogeneous migrations have similar source and target database engines, such as SQL Server to Azure SQL Database. On the other hand, heterogeneous migrations have data transformation steps that convert the original data schemas from the source to the target.

Whatever you do with data, you'd better govern it. That's the topic of our next section.

Governing data

Azure Purview is the last-born data governance tool built by Microsoft for Azure, and Purview is still in preview as of December 2020. It is not exactly a replacement for **Data Catalog**, because both services can live side by side, but it is probably more futureproof. If you start a greenfield Azure data journey, you should investigate Azure Purview.

Azure Purview aims to give you a single pane of glasses to manage your entire data landscape. While the ambition is great, we know that this is very challenging in practice. However, Azure Purview allows you to discover and classify data across on-premises, cloud-based, and SaaS solutions. Beyond data, Azure is releasing more and more cross-cloud services (such as Azure Sentinel, Azure Arc, and so on) that help you govern your hybrid solutions. We do not have a detailed assessment of Purview just yet, but it is certainly worthwhile to look at it.

We have covered our entire data architecture map. Now, let's get our hands dirty with a concrete example, which will let you taste the flavor of a modern data experience.

Getting our hands dirty with a near real-time data streaming use case

The time has come to get more acquainted with a few data services. One of the beauties of the cloud is the possibility to ingest and analyze large amounts of data, which are typically involved in big data scenarios. However, big data services can also handle small amounts of data at a very affordable price. To give you a glimpse of what native Azure services can do for you, we will build a quick solution from scratch, and then we'll see it in action.

Here is the scenario:

We have radars located in different cities, and we want to have a real-time dashboard showing both vehicles going over the speed limit and vehicles observing the speed limit (or less).

To make your life easy, we created a small .NET Core console app that simulates the radars. You can download the app from <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/blob/master/Chapter06/code/devicesimulator.zip>.

But before we start this app, let's first prepare our Azure environment. We will need the following:

- An Azure event hub, which is where our radar simulator will send speed events.
- A **Stream Analytics (SA)** job, which will pull the radar events from our hub. It will calculate aggregates and will route the results to different Power BI datasets.
- A Power BI workspace, where we will build our dashboard.

Because we know data architects may not be very interested in code, we will simply use the portal to provision different services, and we'll use our app to test them. Let's start with the Power BI workspace.

Setting up the Power BI workspace

As stated in the *Technical requirements* section, you must have a Power BI environment. Use a trial version if needed. Here are the steps required to set up our workspace:

1. Navigate to `https://app.powerbi.com/home` and log in.
2. In the menu on the left, click **Workspaces | Create workspace**, and name it `packt`.
3. Navigate to your workspace, and then create an empty dashboard. Our Power BI datasets will be created automatically by SA.

We are done for now. Keep the Power BI page open, as we will need to come back to grant permissions to **Stream Analytics**. Let's now prepare our Azure Event Hub instance.

Setting up the Azure Event Hubs instance

In your Azure subscription, perform the following steps:

1. Navigate to `https://portal.azure.com` and log in.
2. Create or reuse the resource group named `packt`.
3. Inside the **Resource group**, click **New | Event Hubs**, give it a unique name, choose the correct region, and make sure it is well associated with the `packt` resource group. Note that you can select the **Basic** pricing tier.
4. Navigate to your newly created namespace via **Event Hubs**. Add a new hub named `data`, keeping the default options.

Our hub, which will be our input for the SA job, is now ready. Let's finish with the SA job.

Setting up Stream Analytics (SA)

As a reminder, SA will be between our event hub and Power BI. It will read the incoming data stream, apply some queries, and route the results to the Power BI datasets, which will be surfaced on a dashboard. The following steps are required to set up SA:

1. Go to your packt resource group.
2. Add new resource and select **Stream Analytics Job**. Name it packt as well.
3. Once the SA instance is provisioned, go to **Managed Identity**, and check the **Use system-assigned identity** box.
4. Go to **Inputs | Add Stream Input | Event Hub**, and select the hub we created earlier. Name it hub. Make sure to select the **Connection string** option in the authentication mode drop-down list. (The list defaults to **Managed Identity**, but we want to simplify your life for this small exercise, and the connection string option is easier to use.)
5. Before we add our Power BI output, we need to authorize our SA instance to interact with the Power BI workspace that we created earlier. To do so, we first need to allow the use of managed identities. So, go back to the Power BI portal and perform the following steps:
 - a) Click on **Settings | Admin Portal**.
 - b) Go to **Tenant Settings | Developer Settings | Allow service principals to use Power BI APIs**.
 - c) Go back to your workspace and then go to **Access**, search for the SA service name, select the service, and give it the member role. *Figure 6.18* shows you what it should look like:

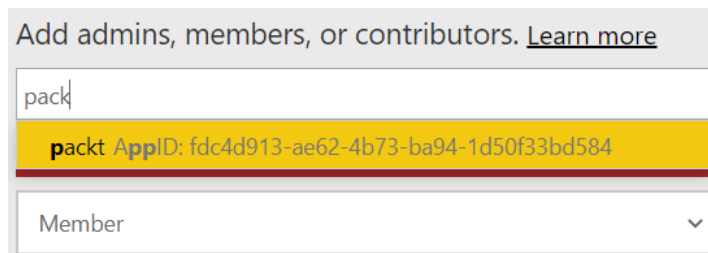


Figure 6.18 – Granting the member role to the SA job

Remember that we enabled the system-assigned managed identity for our SA instance. We now grant that identity access to our Power BI workspace and that is how SA will be able to create its datasets and push data into them.

6. Now, back in SA, you can add the Power BI output. Click **Outputs | Add | Power BI | Authorize**. You will have to log in with your Power BI credentials as an administrator of the workspace. This step will connect our SA instance with Power BI. It will also check whether the identity has access to the target (hence the reason for the previous step). Name your output `powerbi`.
7. Create another Power BI output, as in *step 6*, and name it `fine`.
8. Now that both our input and our output have been defined, you can create the query. Open the query file we provided at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/blob/master/Chapter06/code/sa-query.txt>, and copy its contents.
9. Click **Query**, and then paste the contents into the text placeholder. You should end up with something like the query shown in *Figure 6.19*:

```

1 WITH
2 OK AS
3 (
4     SELECT sensorName, count(*) as TotalUnderMaxSpeed
5     FROM hub
6     WHERE speed <= 50
7     GROUP BY sensorName,
8         TumblingWindow(second,10)
9 ),
10 NOK AS
11 (
12     SELECT sensorName, count(*) as TotalOverMaxSpeed
13     FROM hub
14     WHERE speed > 50
15     GROUP BY sensorName,
16         TumblingWindow(second,10)
17 )
18 SELECT OK.sensorName, OK.TotalUnderMaxSpeed, NOK.TotalOverMaxSpeed INTO
19 powerbi FROM OK INNER JOIN NOK ON NOK.sensorName=OK.sensorName
20 AND DATEDIFF(second, OK, NOK) BETWEEN 0 AND 10
21 SELECT * INTO fine FROM hub WHERE speed > 50

```

Figure 6.19 – Our SA query

As you can see, we have our hub input and our two Power BI outputs on the left. Our job query works with two sub-queries. The first one selects all the measures that are under 50 miles/hour and groups them by sensor. `TumblingWindow` is a contiguous time interval. Our second query selects all the `overspeed` measures. Next, our first main query regroups data from the two sub-queries and sends them to our first Power BI output. The second query sends all the `overspeed` measures to the `fine` output. Let's now see the code in action.

Testing the code

In theory, you should be ready to test the solution. To make it easy for you, we have developed a .NET Core console app, which you can download at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/blob/master/Chapter06/code/devicesimulator.zip>. Since it is an archive, you should unzip it somewhere on your file system. We are going to use a simple MS DOS command prompt to run the application.

Before launching the application, we need to update its configuration file with the connection string of our event hub. To do so, perform the following steps:

1. Locate and open the `appsettings.json` file in the `netcoreapp3.1` folder.
2. Replace your `connection string` with your event hub's connection string. To get this information, do the following:
 - a) Locate your event hub namespace within the Azure portal.
 - b) Click **Shared access policies | RootManageSharedAccessKey | Connection string-primary key**. Copy the value.
 - c) Paste it in the `config` file.

The application is ready to be launched, but let's first have a quick look at its code, which you can find at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/blob/master/Chapter06/code/DeviceSimulatorConsole/Program.cs>.

First, let's have a look at our data object:

```
public class DataObject{
    private string[] sensorNames = new string[] { "Brussels",
        "Genval" };
    public string sensorName { get; private set; }
    public double speed { get; private set; }
    public string plateNumber { get; private set; }
    public DataObject()
    {
        sensorName = sensorNames[new Random().Next(0, 2)];
        speed = (new Random().NextDouble()*100);
        plateNumber = Guid.NewGuid().ToString();
    }
}
```

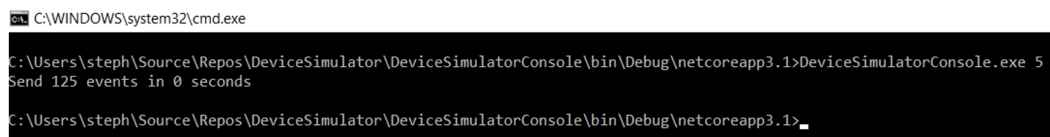
Upon instantiation, we randomly choose a sensor, whose name is the location, and we assign a random speed and a unique plate number.

Here is the body of our main method, which makes use of the data object:

```
await using (var producerClient = new EventHubProducerClient(
    config["EventHubCs"], "data")){
    for (int i = 0; i < instances; i++){
        parallelTasks.Add(Task.Run(async () =>{
            using EventDataBatch eventBatch = await
                producerClient.CreateBatchAsync();
            for (int i = 0;i<25;i++){
                eventBatch.TryAdd(new EventData(Encoding.UTF8.
                    GetBytes(
                        JsonSerializer.Serialize(new DataObject())
                    )))
            }
            Interlocked.Add(ref count, eventBatch.Count);
            await producerClient.SendAsync(eventBatch);
        }));
    }
    await Task.WhenAll(parallelTasks);
}
```

We use the `EventHubProducerClient` and `EventDataBatch` classes to send events to our hub. Each task sends a series of 25 events. The number of tasks is passed as a parameter.

Now you can launch the application by simply running `DeviceSimulatorConsole.exe <n>` as shown in *Figure 6.20*:



```
C:\WINDOWS\system32\cmd.exe
C:\Users\steph\Source\Repos\DeviceSimulator\DeviceSimulatorConsole\bin\Debug\netcoreapp3.1>DeviceSimulatorConsole.exe 5
Send 125 events in 0 seconds
C:\Users\steph\Source\Repos\DeviceSimulator\DeviceSimulatorConsole\bin\Debug\netcoreapp3.1>_
```

Figure 6.20 – Execution result of `DeviceSimulatorConsole.exe`

The `<n>` argument is the number of instances you want to run in parallel. In this example, we ran 5 instances, which produced 125 events and sent them in less than a second. There are a few remaining steps:

1. Start the SA job to extract the events and then send them to Power BI. Locate your SA instance within the Azure portal and simply click **Start**. It may take a few minutes to start.
2. Verify that the two Power BI datasets were created in your workspace. Since you already sent a few events to the event hub and started the job, your datasets should be available.
3. Edit your empty dashboard to add a tile. Choose the **Custom Streaming Data** tile type.
4. Choose your powerbi dataset | **Clustered column chart** | `sensorName` in the axis. Choose `TotalUnderMaxSpeed` and `TotalOverMaxSpeed` in **Values**. Choose a frequency of **1 minute**.

After a few seconds, you should see your dashboard, as shown in *Figure 6.21*:

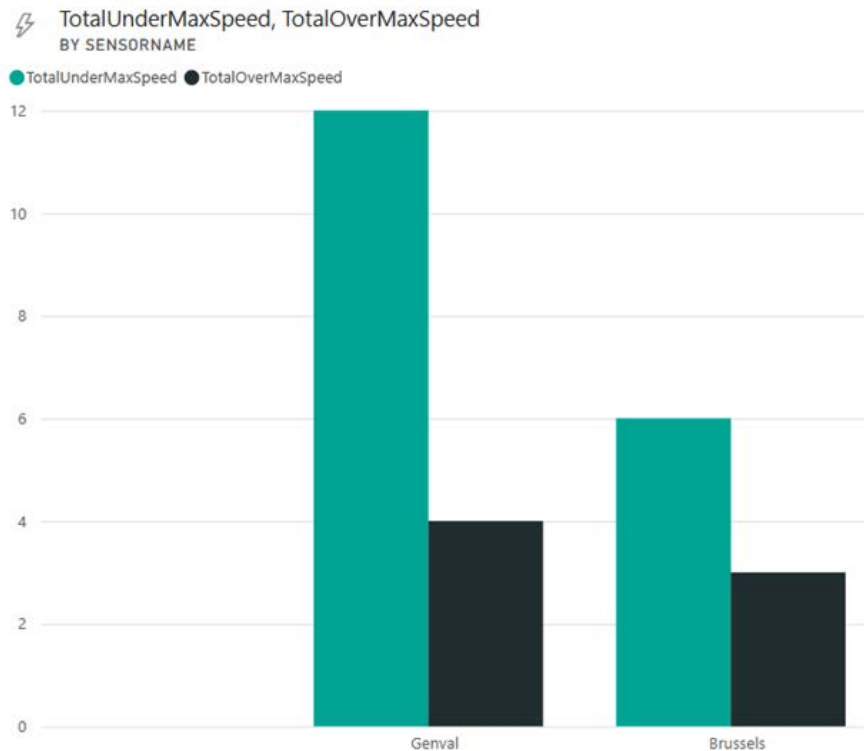


Figure 6.21 – Real-time dashboard showing the speed of vehicles

If you do not see anything, refresh the page.

Remember that we have two datasets: `powerbi` and `fines`. If you explore the data in `fines`, you should see all the vehicles that were caught driving over the speed limit, as shown in *Figure 6.22*:

< Back to report

plateNumber	sensorName	speed
084647c6-c4ed-40b4-9953-775035319263	Brussels	80.33
49944c1b-b0f2-4996-a60a-207c900ee610	Brussels	87.47
9cee2149-81b3-46d5-bc85-990c0b05d735	Brussels	77.56
0cfc2f6f-6bc0-41a1-b330-76b2a7e75dd2	Genval	99.01
50e7172f-169d-4447-8f74-7947076fbd36	Genval	62.34
a1d6fcb0-3953-4c85-a687-cd5452d1e4f5	Genval	67.80
de07ade0-7ca1-409f-8c95-982970684fcb	Genval	74.21

Figure 6.22 – Raw data report of flashed vehicles

In our example, we simply sent that data to a Power BI dataset, but you could send it to a fine-recovery service to undertake concrete actions. We also dealt with a very limited number of events, but we could easily scale this by using auto-scaling for SA. Let's now recap the chapter!

Summary

In this chapter, we browsed the vast data landscape of Azure. We split the traditional and modern data technologies to help you understand the possible solutions should you be on the verge of a transition from regular BI to advanced analytics, or from ELT to ETL.

Azure shines in the big data space, and that may be the area where only public cloud providers can make a real difference compared to on-premises systems. We showed you how Azure has both a native big data offering as well as solutions that are open source in origin. Many services are intermingled, but the trend is to let ASA be at the center of everything.

Lastly, we worked on a concrete hands-on exercise to give you a glimpse of what a modern solution might look like. We completed an end-to-end scenario, from data ingestion and real-time queries to surfacing the results on a real-time Power BI dashboard. You should now have a better understanding of when to use what, and how to get started with top-notch data services.

Our next chapter is about security in Azure. The public cloud, only by the mention of its name, often agitates the traditional security architects. Let's see what modern security looks like and how to give peace of mind to our blue and red teams.

7

Security Architecture

In this chapter, we emphasize and explain the importance of security in the cloud. We will explore security architecture, explaining the paradigm shift in identity in the cloud. Finally, we will drill into several use cases in order to show the practical applications of our recommendations.

We will more specifically cover the following topics:

- Introducing cloud-native security
- Reviewing the security architecture map
- Delving into the most recurrent Azure security topics
- Adding the security bits to our Contoso use case

By the end of the chapter, you will have a better understanding of cloud-native security and a better knowledge of recurrent and typical Azure security topics.

Let's begin by reviewing the technical requirements.

Technical requirements

There are no hands-on exercises, so there are no specific technical requirements. All the diagrams and maps are available (in full size) at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/tree/master/Chapter07>.

First, let's introduce cloud-native security.

Introducing cloud-native security

In light of what we have seen so far in previous chapters, we know that the cloud can help us develop and deploy solutions faster and at a better cost. However, that is only true if we also modernize the way that we secure our workloads. Cloud-native security relies on the **Shift-Left** principle, which consists of integrating security processes earlier in the life cycle of an asset. Considering security from the ground up prevents unexpected delays and surprises later, prior to the production deployment. However, this is easier said than done!

Often, we see developers (usually early adopters) and infrastructure engineers embracing this modernized way of working (with **Infrastructure as Code (IaC)**), while security remains organized in a traditional way (waterfall and reactive). Often, you must wait weeks, if not more, to have a firewall rule ticket request accepted and implemented. This way of working is the exact opposite of the Shift-Left mindset, and it tends to annihilate the efforts of other teams.

With the proper tooling and technologies in place, security becomes declarative. In *Chapter 4, Infrastructure Deployment*, we saw how CI/CD factories help organizations to provision both code and infrastructure. The same concept applies to security, even more with true cloud-native platforms, such as Kubernetes, where network policies, ingress rules, and so on are all automated. Also, cloud-native often means polyglot applications, made of multiple programming languages and a ton of open source libraries. That is why integrating security in your CI/CD factory is even more important.

However, cloud-native security is not just a technology matter; it is a *mindset*. It requires different organization and a different culture, which is much harder to achieve since organizations often prefer decades of legacy practices. Whatever industry you are in and whatever regulatory obligations you have, there is always room for automation and more fluent processes. Regulators often define the what, not the how!

The cloud-native security mindset implies that security architects should create value for the organization, and they should find the means to automate security practices and ensure their processes do not become an impediment. They should be enablers, not disablers or showstoppers. They should help non-security people realize why security is important and bring them solutions. More importantly, they should make risk assessments and let the business decide on the residual risks they are willing to accept.

This mindset shift is not easy to achieve, especially in some industries, such as banking, whose DNA is naturally risk-averse. The extent to which you will be able to transition from traditional security to cloud-native partially depends on this DNA. But make no mistake: while the cloud and related technologies can really make a difference, you will *never* achieve the promises (cost-friendliness, high velocity, robust and scalable solutions, and so on) if you do not change your security practices. Security is an integral part of that success or failure.

From a technical perspective, there is also a shift toward identity as a primary layer of defense, instead of the network, although Azure has filled many feature gaps in the network area over the past years. Looking back, in 2015, Azure did not have many network features, other than for pure **Infrastructure as a service (IaaS)** workloads. In 2020, Microsoft largely expanded its network-related features for its **Platform as a service (PaaS)** and **Function as a service (FaaS)** offering. However, if you rely on the network as a primary layer, it often remains challenging, more expensive, and sometimes even convoluted. This technical shift is often not well understood by traditional security architects, who too often resort to network-only security (the perimeter obsession and the DMZ mindset) while neglecting the identity aspect of security.

Throughout this chapter, we will try to distill some cloud-native advice and draw your attention to typical, traditional practices that might represent an impediment to a smooth cloud journey. Remember that the success of this journey is tightly coupled with your security practices.

Let's now review the security architecture map.

Reviewing the security architecture map

In this section, we will browse the main security-related services, with a special focus on identity, the cloud's primary defense layer. Our objective is to make you realize the importance of identity in Azure. We already covered most of the network plumbing in *Chapter 3, Infrastructure Design*, so we will now essentially review some service-specific network features. We will also look at the various encryption possibilities, and more globally, how to handle your security posture. *Figure 7.1* shows the security areas that we will explore:

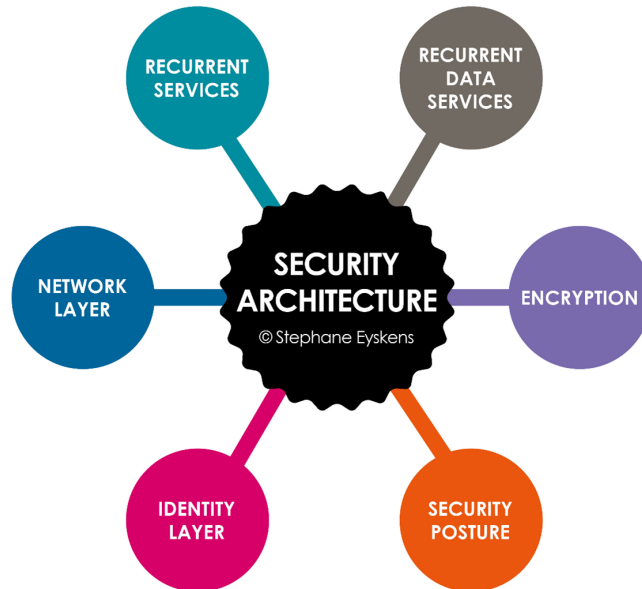


Figure 7.1 – The security architecture map

Important note

To see the full security architecture map (*Figure 7.1*), you can download the PDF file at <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook/blob/master/Chapter07/maps/Security%20Architecture.pdf>.

Our map has six top-level groups:

- **RECURRENT SERVICES:** We will see the different service-level options that we can use as part of our security arsenal.
- **NETWORK LAYER:** This layer is there for the sake of completeness, but it is essentially similar to the network layer that is part of the infrastructure architecture map, since it mostly focuses on bridging data centers (and it depicts the hub and spoke topology). We will not repeat what we told you previously. Instead, we will highlight some network-related trade-offs that we will tackle in the upcoming subsections.
- **IDENTITY LAYER:** As stated in the section introduction, we will take an in-depth exploration of the identity piece, our primary layer of defense in the cloud.
- **RECURRENT DATA SERVICES:** We will explore the data-specific security features.
- **ENCRYPTION:** Encryption also has a strong emphasis on the cloud, so we will discuss **Bring Your Own Key (BYOK)**, **Hold Your Own Key (not Host)**, and **Service-Managed Key (SMK)** concerns.
- **SECURITY POSTURE:** We will explore some built-in tools that help you to manage your security posture, as well as see how to integrate them with on-premises systems.

Before we dive into more specific areas of the map, let's give you a brief description of a few topics that will be recurrent across multiple top-level groups, and that we will analyze further in the *Delving into the most recurrent Azure security topics* section:

- **Azure Private Link (APL):** In a nutshell, APL allows us to define a private endpoint for a public PaaS service, to prevent internet inbound traffic.
- **Shared Access Signatures (SASes):** In a nutshell, SASes are a historical way to authenticate against various services, such as Azure Storage, Azure Service Bus, IoT Hub, and so on. It is gradually being replaced by **Azure Active Directory (AAD)**. However, SAS is still heavily used.
- **Managed identities:** These represent an Azure-only method of authenticating against a resource, by using AAD. Managed identities can replace SAS wherever possible.

With this clarification made, let's start with the **RECURRENT SERVICES** group.

Exploring the recurrent services security features

In this section, we will explore the security features of the most frequently used services. The groups are highlighted in *Figure 7.2*:

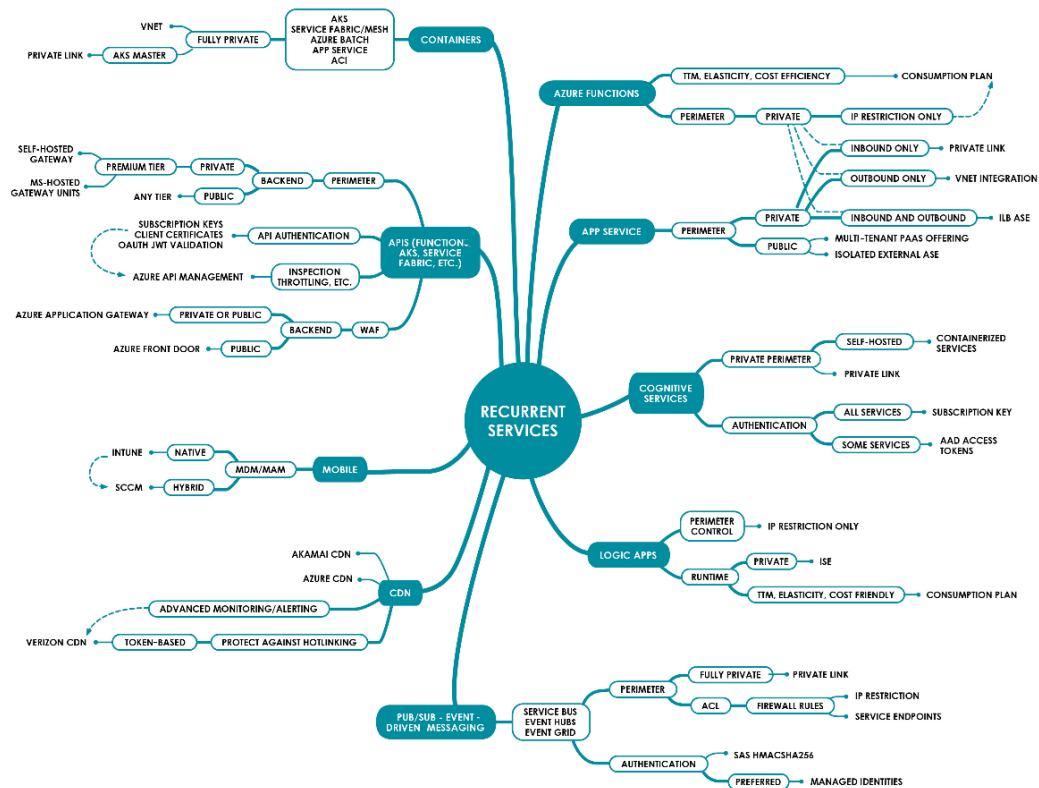


Figure 7.2 – The recurrent services security features

Let's now explore the first group, the API-related security features (at the upper left of *Figure 7.2*).

API security features and topologies

For every API workload, and mostly APIs that are exposed to external parties (B2B and B2C), you will leverage Azure **API Management (APIM)**. If the backend services, proxied by APIM, are in a private perimeter, you will have to use the premium pricing tier, which gives you the following possible options:

- **Use Microsoft-hosted gateways:** Each gateway unit costs about \$2,700/month. The advantage is that they are entirely managed by Microsoft. They also support geo-redundancy for global API deployment.

- **Use self-hosted gateways:** Each self-hosted gateway costs about \$800/month. The cost savings are substantial, but the primary reason why you would self-host a gateway is for multi-cloud and/or hybrid solutions. For example, when your backend services are on-premises, a self-gateway unit is a better option than a Microsoft-hosted gateway, in terms of data-in/data-out. Of course, with self-hosted units, you are entirely responsible for the high availability and disaster recovery mitigation.
- **Use both at the same time:** You can combine self-hosted units with Microsoft-hosted units, for instance, in hybrid workloads where some backend services are in the cloud and others are on-premises.

For backend services that would also be publicly accessible, you can use any pricing tier. For example, you can have a backend hosted on an Azure app service or as an Azure function on the public PaaS offering, as shown in *Figure 7.3*:

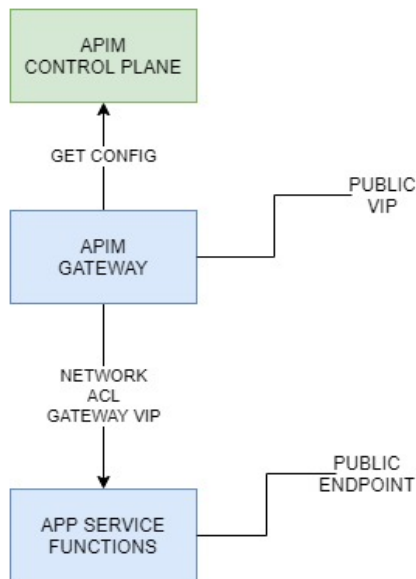


Figure 7.3 – APIM and public backend

IP restrictions are set at the app service (or function app) level, to only allow calls coming from the APIM gateway, the **Policy Enforcement Point (PEP)**. To ensure a **Web Application Firewall (WAF)** feature, the gateway itself can be proxied by an Azure Application Gateway or Azure Front Door instance, depending on the gateway VIP type (public or private). Azure Front Door is currently limited to public backends only.

In terms of policies, APIM allows you to control almost everything, thanks to its rich policy engine. The typical policies that you can enforce are as follows:

- JWT validation, to make sure that every request is authorized to connect to the underlying backend. You can connect APIM to any **OpenID Connect (OIDC)** IDP, providing its discovery endpoint (`/.well-known/openid-configuration`) is connectable. If the IDP cannot be connected, its keys can be registered in APIM directly. JWT validation typically involves the validation of the token issuer, the audience, and some extra claims, such as scope validation.
- **Mutual TLS (mTLS)** through client certificates. This is one of the built-in APIM policies.
- Subscription keys can be issued by the product or by an API. Subscription keys are, in theory, more suited for usage reporting, instead of pure authentication. However, in pure B2C scenarios, where you do not want to impose too many constraints on the subscribers, the keys still represent a way to identify a subscriber and to protect an API.
- Among various other policies, throttling prevents the abuse of the API. Throttling represents an effective DoS/DDoS mitigation, because every attempted instance of abuse (or incorrect request) is discarded by the gateway.

Let's now look more closely at Azure App Service and Azure Functions, which are often proxied by APIM.

Exploring Azure App Service and Azure Functions

In this section, we are going to explore two of the most frequently used Azure services, namely Azure App Service and Azure Functions. *Figure 7.4* shows the various options at our disposal:

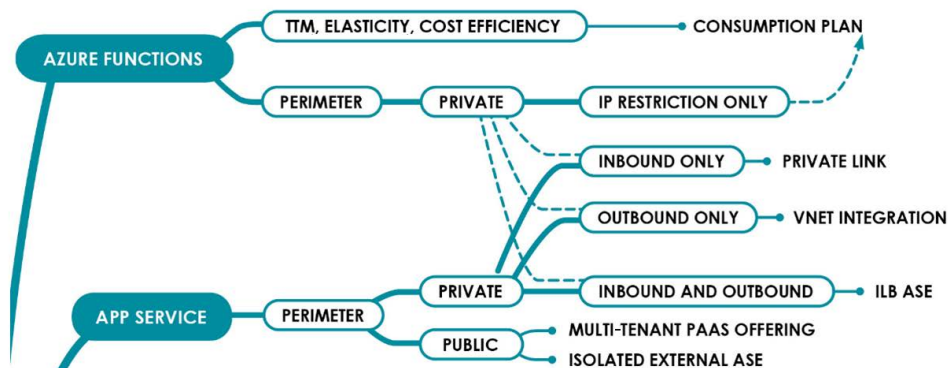


Figure 7.4 – Azure App Service and Azure Functions security features

Azure Functions relies on the same building block as Azure App Service, with one noticeable exception: the **consumption tier**, which is also known as the **serverless tier**. Other than the serverless tier, both Azure Functions and Azure App Service are built on top of an **App Service plan**, which defines the prepaid compute. A plan can host one or more apps and can run one or more instances. The total cost equals the plan price times the number of instances.

The consumption plan is by far the cheapest (free: 1 million calls/month, about 50 cents for every extra million calls), the most elastic (scales out automatically), and the fastest option to launch something in production, because there is nothing you have to do but deploy the function code. However, from a perimeter perspective, the function runtime runs in a public Microsoft-controlled network perimeter. Azure Functions is hosted on function apps, which support network **Access Control Lists (ACLs)**, in order to restrict inbound access to some IP ranges, such as the VIP of an APIM gateway.

If you want to limit the exposure even more, you can use APL, but this is only available as part of the premium tier. If your function or application must connect to a resource that is inside a VNet, you can leverage the VNet integration, which is a way to redirect outbound traffic through a VNet, while the service instance itself lives *outside* the VNet.

Another possibility is to leverage the fully isolated flavor: the **App Service Environment (ASE)**. With an ASE, the app service(s) is fully inside a VNet. Both inbound and outbound traffic can be controlled using **Network Security Groups (NSGs)** and **User-Defined Routes (UDRs)**, to an **Network Virtual Appliance (NVA)** or Azure Firewall. The ASE is by far the most expensive and complex option, but it is fully in line with the hub and spoke topology. When opting for an ASE, you must also purchase the **Isolated App Service plan**. The ASE is available in the following two modes:

- **The internal load balancer (ILB) App Service environment (ASE)**: This option removes every public endpoint and is **Payment Card Industry (PCI)**-compliant.
- **The external ASE**: This option still exposes public endpoints while automatically having access to resources that are inside a VNet, and it is not hosted on a multi-tenant offering.

Both ASE options are quite expensive. However, Microsoft launched a preview of ASE v3 in November 2020. ASE v3 is announced to be simpler and cheaper than its predecessors, because Microsoft has decided to eliminate the ASE flat fee cost, which was about \$1,000/month. Therefore, costs drop by about 80%. In any case, if you opt for an ASE, you should consider deploying more than a single application to amortize the costs incurred by the ASE flat fee (if the version is before v3) and the isolated App Service plan (up to \$850/month/instance).

At last, another way to privatize Azure Functions is pack them as containers and host them in any orchestration platform that you control. Of course, this makes you responsible of the availability and disaster recovery bits.

You must pay special attention to Azure Functions for any network restriction that you put in place, because they have bindings and triggers. As we saw in *Chapter 5, Application Architecture*, most modern applications are distributed and event-driven. At the time of writing, Azure Functions instances that do not have a public endpoint cannot be called by Azure Event Grid, because it only supports public endpoints. Thus, the scenario shown in *Figure 7.5* is impossible:

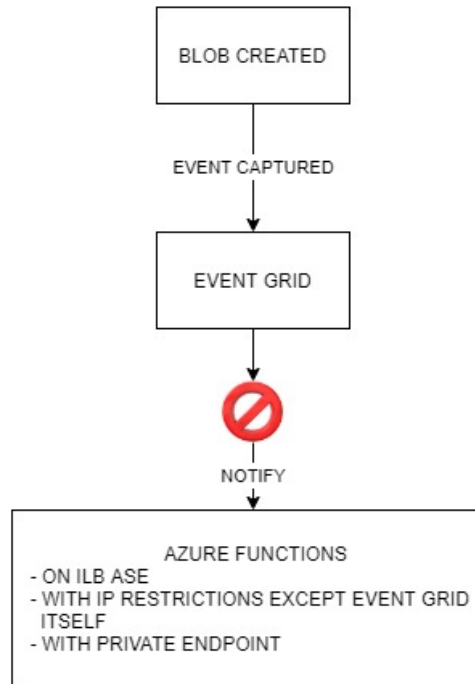


Figure 7.5 – Network impact on Azure Function triggers

The same could apply to the HTTP trigger, depending on where the caller is. A mitigation to this problem is to place an Azure service bus in between Event Grid and Azure Functions, as shown in *Figure 7.6*:

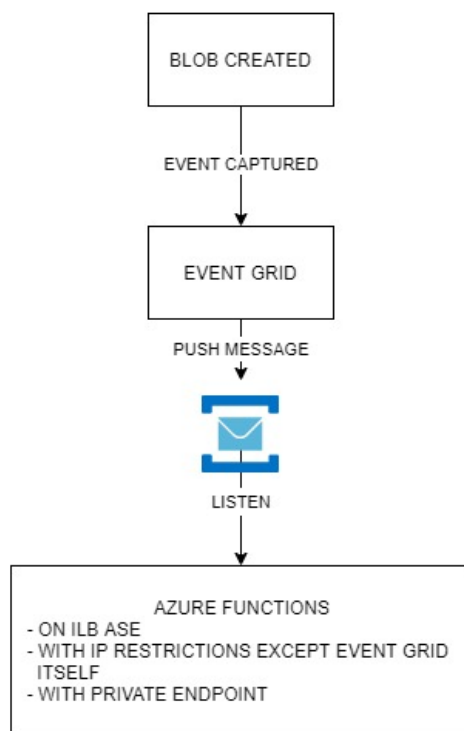


Figure 7.6 – Network impact mitigation with Azure Service Bus

If you let Azure Event Grid send events to Azure Service Bus, the receiving function can connect to the bus to read incoming messages. However, you end up with the following:

- You still have a public endpoint that is the bus itself.
- You add an extra service only to satisfy perimeter requirements. This generates extra costs and extra complexity.

A variant of the preceding is to proxy your Azure Functions with APIM, but here, again, you have an extra service.

So, you should think twice before enforcing network restrictions in one way or another in Azure Functions. The latter example naturally leads us on to our next topic, pub/sub and EDA services.

Security with pub/sub and EDA services

Given the distributed nature of cloud and cloud-native applications, Azure Service Bus, Azure Event Hubs, and Azure Event Grid are certainly part of 80% of applications. *Figure 7.7* shows you how to control the perimeter and how to authenticate against these services:

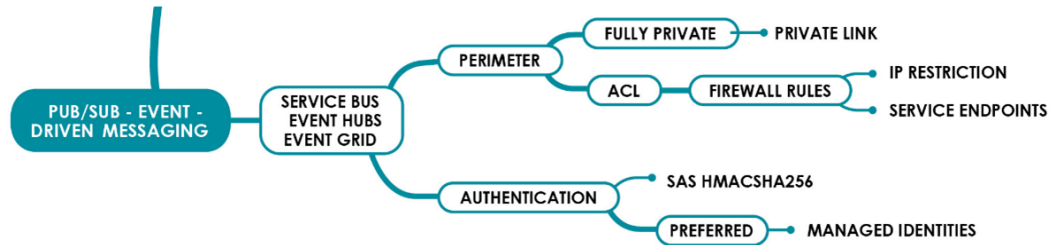


Figure 7.7 – Securing pub/sub and EDA services

From a perimeter perspective, you can have a fully private **SERVICE BUS, EVENT HUB,** or **EVENT GRID** using APL. Their public endpoints are all proxied by a firewall, which allows you to define both IP restrictions and **Service Endpoints (SEs)**. Whether your instance is public or private, you can authenticate using a **SAS** or using **AAD** via **managed identities**, if the client is hosted on Azure. Or, you can use regular AAD token requests if the clients are hosted on-premises or in another cloud. We will explain APL, SEs, and managed identities in detail later in this chapter.

Let's now take a look at various other usual suspects that are part of many Azure solutions.

Exploring other recurrent services

A **Content Delivery Network (CDN)** is used to speed up static content delivery to client devices. The delivered contents are often accessible anonymously. Indeed, static contents (such as JavaScript files, images, and so on), which comprise a user interface, are usually not highly sensitive. If you do not have any special requirements, you can use any Azure CDN service. However, should you require specific conditions, you can use **token authentication**, but it is only available with **Azure CDN Verizon Premium**. Token authentication allows you to define a series of rules that are encrypted with a key defined, and the rules are stored in the CDN management console. Such rules can be a combination of HTTP headers, the IP origin of the caller, and so on.

Any access to the CDN requires the token to be part of the query string. Should a link be leaked and used from another context that is not in line with the defined rules, then access to the resource will be denied by the CDN service. Even with anonymous content, Verizon CDN helps fight against **hotlinking**, which is a technique used to abuse CDN instances. While CDN architectures are built to be robust and resilient against DoS/DDoS, you are charged for the bandwidth costs, which you could avoid with token authentication.

When it comes to mobile apps, you can rely on **Intune**, with or without **System Center Configuration Manager (SCCM)**. Intune can be used for both **Mobile Device Management (MDM)** and **Mobile Application Management (MAM)** purposes. With MDM, you can verify that client devices belong to your company, as well as to possibly enroll personal devices (**BYOD – Bring Your Own Device**). With MAM, you can define application-level policies, as well as application-level data wiping. For example, you can perform tasks when a device is lost or stolen, or when employees leave the company and have employee-owned devices with corporate apps.

APL can also be used with some **Azure Cognitive Services (ACS)** instances, while some services can also be self-hosted as containers in a private perimeter of yours. With regards to authentication, all ACS instances support key-based authentication through the subscription key, while some of them also support AAD-based authentication.

The entire container stack supports VNet encapsulation, and the **Azure Kubernetes Service (AKS)** master API can be private link-enabled, so as to end up with a fully private AKS.

At last, **Azure Logic Apps** supports network ACLs and can be proxied by APIM. Note that like Azure functions, logic apps also have triggers, so the potential problems we highlighted earlier also apply here. Independently of the inbound traffic, the Logic Apps runtime can be serverless or self-hosted (that is, an **Integrated Service Environment (ISE)**). Here, again, we faced the same considerations that we had for Azure Functions. It's worth mentioning the relatively high cost of an ISE, which is about \$4,700/month, where once more the serverless tier is very cheap and adjusted to the actual consumption. So, again, you should pay special attention when privatizing logic apps.

Let's now look at the recurrent data services.

Exploring the recurrent data services security features

In this section, we will see the most frequent data services that are used in probably 80% of solutions. *Figure 7.8* shows that Azure Storage and PaaS databases are part of these services, while **Azure Data Factory (ADF)** is often used to import blobs from Azure Storage and write them to PaaS databases (or other systems):

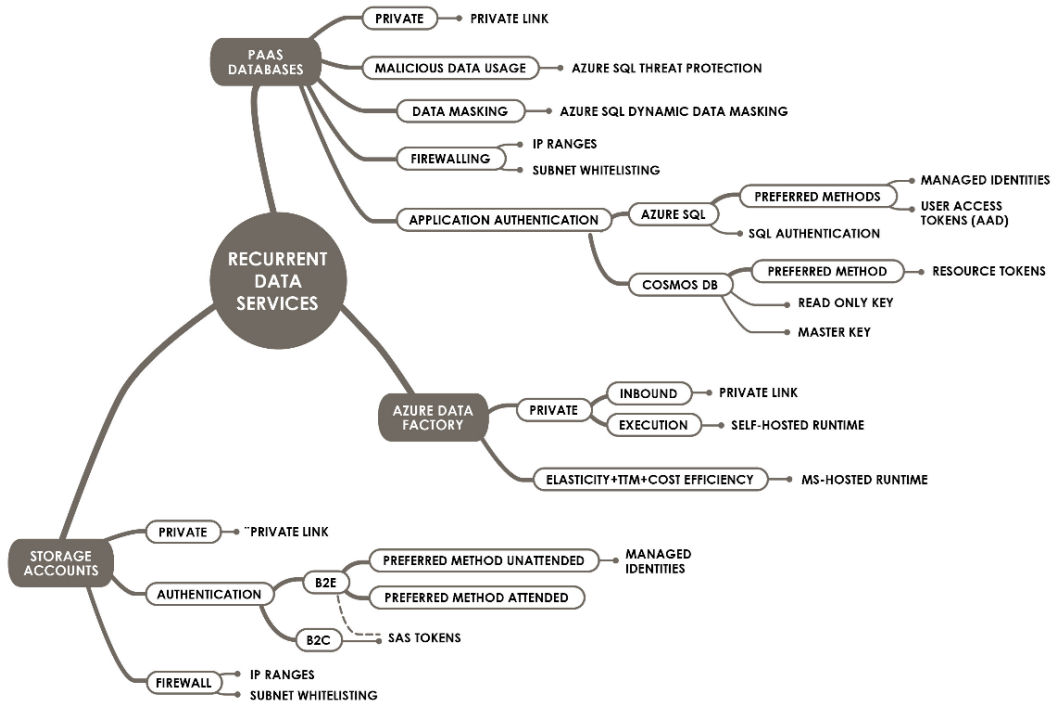


Figure 7.8 – Securing some recurrent data services

All PaaS databases can now be fully privatized, thanks to the private link feature, which we will explain later in this chapter. The SQL stack (including Azure SQL Database, SQL Managed Instance, and Azure Synapse Analytics) can be closely monitored by **Advanced Threat Protection (ATP)**. ATP is a service that helps you detect (and respond to) security incidents. It is basically also part of your security posture, since it integrates with **Azure Security Center (ASC)** (which we will discuss later).

Azure SQL has a **Dynamic Data Masking** feature that prevents disclose-sensitive information. For example, a credit card number would be masked with asterisk characters (in the result of a query) to avoid a service administrator from being able to see all the digits. Masking relies on masking rules. Ordinary database users will always see the masked values, unless they are excluded from the rule. Database administrators always see the unmasked values. In terms of a firewall, a PaaS database supports IP ranges and subnet whitelisting. We will explain resource firewalls, beyond PaaS services, in the *Delving into the most recurrent Azure security topics* section.

In terms of authentication, Azure SQL supports the native SQL authentication mode, but this is largely superseded by AAD authentication. AAD authentication is done through managed identities for service-level database access, while user-based access can be done with user access tokens. This means that user-level security trimming can be enforced in the database itself. As of December 2020, Cosmos DB does not support AAD authentication. An alternative is to use managed identities to grant access to the Cosmos DB keys. Then, you would authenticate to Cosmos DB using the master key, a read-only key, or resource tokens. Resource tokens represent the most granular way of securing Cosmos DB. We recommend that you use resource tokens over any other method in a *least-privilege principle* approach.

ADF also supports a private link for its inbound endpoints. You should not be confused with its runtime, which is the process that reads and writes from/to data sources. Therefore, it won't help to enable a private link to access private data sources. The ADF runtime hosted by Microsoft is called **auto-resolve** and is by far the cheapest, easiest, and fastest way to get started with ADF. However, one downside is that you do not have control over its network perimeter, so you cannot restrict PaaS databases to your ADF instance only. Azure Storage considers ADF as a trusted service, but it is the only one so far among the resource firewalls. This lack of perimeter control often pushes companies to self-host the runtime, in order to restrict the data source access. Self-hosting the runtime might also help you reach out to private data sources. At the time of writing, the ADF runtime ships as a Windows service, so you need at least one **Virtual Machine (VM)** to host it in production. (You should use two VMs for high availability.) Of course, self-hosting the runtime has an impact on the elasticity, costs, and time-to-market.

A more recent possibility, that is still in preview in January 2021, consists of using the auto-resolve runtime with the managed virtual network option. This allows you to connect to PaaS services sitting behind private endpoints. At this stage, you still can't resolve on-premises endpoints but it is a very valid option for Azure-only workloads.

Lastly, the Storage account service is probably the most flexible service, because it supports private link, AAD, and SAS authentication.

Let's now zoom in on encryption.

Zooming in on encryption

In this section, we will explore the different encryption possibilities, as illustrated in *Figure 7.9*:

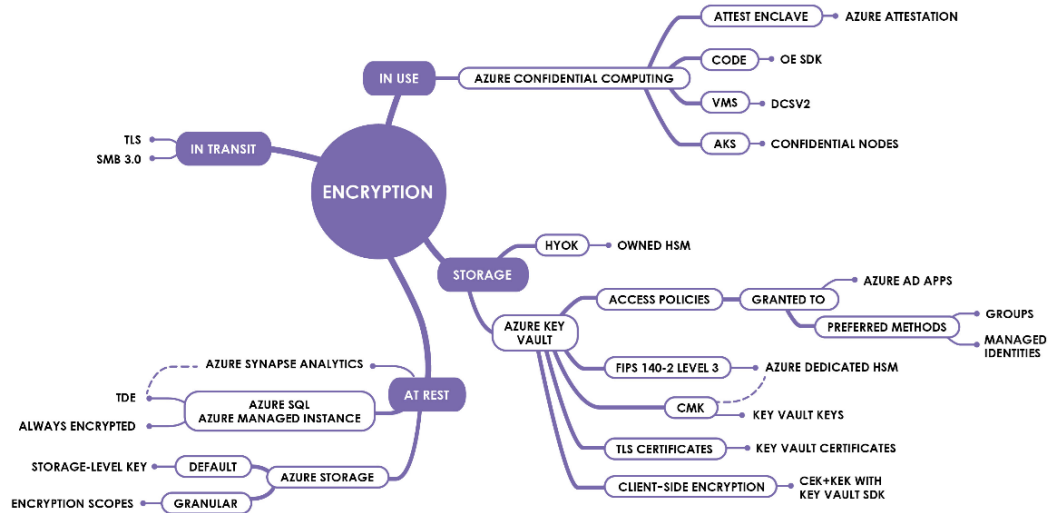


Figure 7.9 – Encryption in Azure

Azure Key Vault is a central component to store keys, secrets, and certificates. We will hold off on the **HYOK** option, which consists of using an owned **Hardware Security Module (HSM)** to store keys. HYOK is mostly used with **Azure Information Protection** in the context of Office 365, but it is incompatible with most PaaS and FaaS services. Therefore, we will not explore it further, but at least you know that it exists. This leaves us with **BYOK**, which is also referred to as **Customer-Managed Keys (CMKs)**, and **SMKs**, which are built-in Microsoft keys. Both encryption modes rely on Azure Key Vault. Again, you have the following two choices:

- Use the multi-tenant Key Vault offering. This is the out-of-the-box option. It's the cheapest and the fastest but you must trust Microsoft (to provide this service). You also must be under regulations that allow the use of *FIPS 140-2 Level 2* HSM.

- Use an Azure dedicated HSM. This is an isolated, single-tenant offering with *FIPS 140-2 Level 3* compliance. The major difference between a dedicated HSM and a multi-tenant offering is that Microsoft does not have administrative control over the dedicated HSM. While this is a good fit for pure IaaS workloads, it is incompatible with many important PaaS services (which include Azure SQL, Azure Storage, Cosmos DB, and so on). This makes it hard to use in practice.

A key lesson learned from the field is that unless you really have a very strict regulation that forces you to use a dedicated HSM, you're better off trusting Microsoft and using the multi-tenant offering of Key Vault (or stay on-premises if you do not trust the platform and its vendor). Any other option would seriously hinder your Azure journey.

Now that that's been clarified, let's explore the different types of encryption.

Encryption in transit and client-side encryption

Encryption in transit consists of encrypting data as it moves. The most common example is when a browser connects to a server over HTTPS. For encryption in transit, you may enforce TLS with all Azure services. As of December 2020, most services are still on TLS 1.2. For public certificates, Azure Key Vault integrates with DigiCert and GlobalSign to let Key Vault generate and rotate certificates automatically. Alternatively, you can make a separate certificate creation process, and then import them into Azure Key Vault.

On top of TLS, Azure also makes use of SMB 3.0 for Azure Files. Other than for deploying web apps, there is no built-in support of FTPS or SFTP, but there are some marketplace offerings. Microsoft also proposed an interesting alternative (<https://docs.microsoft.com/samples/azure-samples/sftp-creation-template/sftp-on-azure/>) using Azure Container Instances. For client-side encryption, which consists of encrypting data on the sender's side, you can also leverage Azure Key Vault, which has special APIs for key operations. More specifically, you can wrap the **Content Encryption Key (CEK)** with a **Key Encryption Key (KEK)**.

We will now look at encryption at rest.

Encryption at rest

Encryption at rest consists of encrypting *stored* data. The purpose is to avoid disclosing sensitive information if unexpected (logical or physical) access to the data store occurs. A popular example of this is Microsoft's **BitLocker**, a full-volume encryption that's based on the **Advanced Encryption Standard (AES)** encryption module. Azure SQL and Synapse Analytics make use of **Transparent Data Encryption (TDE)**, which supports both SMK and CMK. When using SMK, TDE is enabled by default. When using CMK, you must first generate and store your key, and then assign the key to the TDE engine. If you plan to use CMK, try to do it from the start, because switching from CMK to SMK might not always be possible. *Figure 7.10* illustrates encryption at rest:

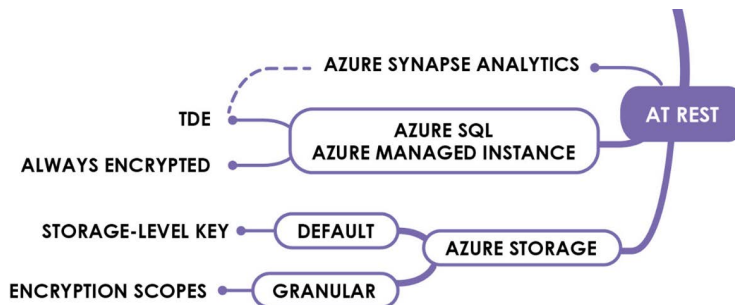


Figure 7.10 – Encryption at rest

Azure SQL and SQL Managed Instance also support **Always Encrypted**, which can be used together with TDE. Always Encrypted leverages client-side encryption so that only an encrypted **Data Encryption Key (DEK)** is stored inside the database. The KEK used to encrypt the DEK is stored outside the database, such as with Azure Key Vault, and it is made available to the Always Encrypted service. The purpose of Always Encrypted is to prevent database administrators from viewing sensitive data. Additionally, Always Encrypted enables encryption in use, which we will tackle in the next section.

All the other data services have their own encryption at rest mechanism, and most of them can be used with CMK. It's worth mentioning that Azure Storage can use storage-level keys (by default), as well as **encryption scopes**, which are still in preview (as of December 2020). Encryption scopes give you more granular encryption by leveraging blob- or container-level keys. The primary use case is to store multi-tenant data in a single storage account. Of course, this could also be achieved with different storage accounts, providing the number of customers is limited.

We will now explore encryption in use.

Encryption in use

Encryption in use mostly consists of encrypting in-memory data. It is part of a broader **Trusted Computing** initiative, which is also referred to as **Confidential Computing**. It complements encryption in transit (data in movement) and encryption at rest (stored data). Both are very handy, but they leave in-memory data unencrypted, making it vulnerable to server attacks (such as memory dumps). Encryption in use relies on **Trusted Environment Execution (TEE)** and is a hardware-level encryption technique. TEE ensures application enclaves where the code executes. *Figure 7.11* shows the Azure confidential computing landscape, which will probably grow over time:

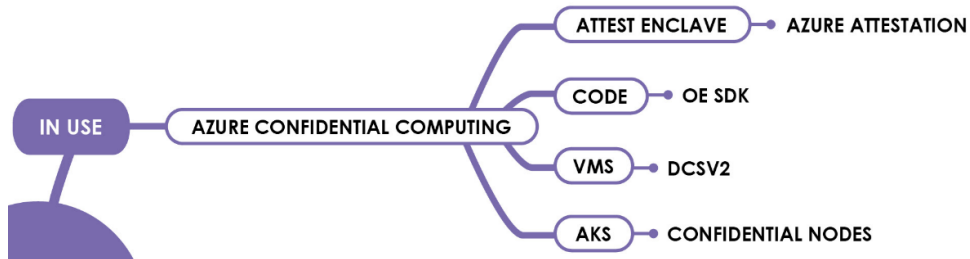


Figure 7.11 – Encryption in use

Confidential computing is rather recent in Azure (most services are still in preview as of December 2020). Nevertheless, since this encryption technique is based on specialized hardware, Azure ships the **DCsv2** VM series. The same node type can be used with AKS node pools, which brings **confidential containers** to AKS. Microsoft partners with third parties to enable the containers. The **Open Enclave (OE)**; (<https://openenclave.io/sdk/>) SDK lets C and C++ developers write custom code that deals with enclaves and TEE. Finally, Azure Attestation is a cross-enclave and multi-tenant service that helps validate that the enclaves are genuine.

Managing your security posture

Let's first define what security posture means. **Security posture** refers to the company's overall cyber resilience against adverse events. This goes far beyond the cloud, because it also applies to on-premises environments, which is where most enterprises still have a larger footprint. Azure comes with many different services to help manage Azure components, as well as on-premises components. *Figure 7.12* shows some of the Azure services that can empower your security posture:

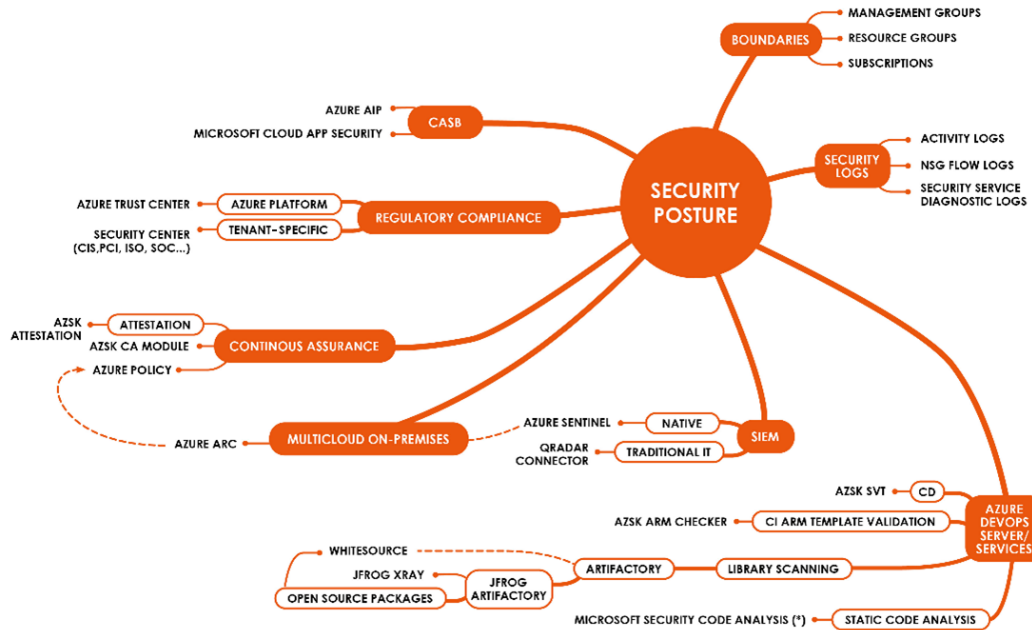


Figure 7.12 – Security posture in Azure

Let's start with on-premises solutions, such as **QRadar**, which is often required to integrate. The integration pattern is almost always the same, with whatever on-premises solutions you must integrate with (QRadar, Splunk, and so on). You redirect Azure logs to Azure Event Hubs and let the on-premises solutions ingest these logs. As an Azure architect, this can greatly simplify your life, because you may argue that your job is done!

It's then up to the internal **Security Operations Center (SOC)** to build the scenarios to detect and respond to any security incidents. Nevertheless, because you are a conscientious professional, you do not want to discard native services just like that, and that is why we recommend you to enable **Azure Defender** in ASC as the bare minimum. ASC has free basic coverage, which will highlight some potential security vulnerabilities of your Azure environment, but Azure Defender covers a broader set of resources. Before you enable Azure Defender, you can see the types of resources and the associated costs in *Figure 7.13*:











Total: 84 resources			
	10 Servers	\$15	Server/Month
	1 App Service instances	\$15	Instance/Month
	1 Azure SQL Database	\$15	Server/Month
	52 Storage accounts	\$0.02	10k transactions
	10 Kubernetes cores	\$2	VM core/Month
	2 Container registries	\$0.29	Image
	8 Key Vaults	\$0.02	10k transactions
	0 SQL servers on machines ⓘ	\$15	Server/Month
		\$0.015	Core/Hour
	Resource Manager (Preview)	FREE during preview	ⓘ
	DNS (Preview)	FREE during preview	ⓘ

Figure 7.13 – Azure Defender

Of course, costs vary according to the number of resources you have. You should at least activate Azure Defender for a trial period to see its added value.

The biggest advantage of ASC is that you have nothing to do but look at its various dashboards, and then analyze the recommendations. For example, *Figure 7.14* shows the overall compliance of an Azure environment with well-known security standards and frameworks:

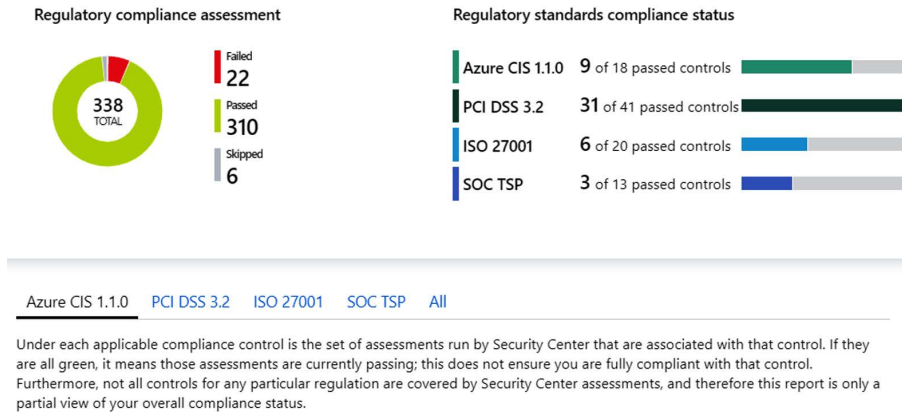


Figure 7.14 – ASC compliance dashboard

As you can see in *Figure 7.14*, your environment is validated against **CIS 1.1.0**, **PCI**, and security standards, which are all baked into ASC. Note that Azure does not cover these standards entirely. Additional standards can be added, thanks to the dynamic compliance packages (<https://docs.microsoft.com/azure/security-center/update-regulatory-compliance-packages>).

ASC will also highlight resource-level security issues and will guide you to a remediation process, or it will offer to remediate the problem for you automatically. *Figure 7.15* shows such resource-level issues:

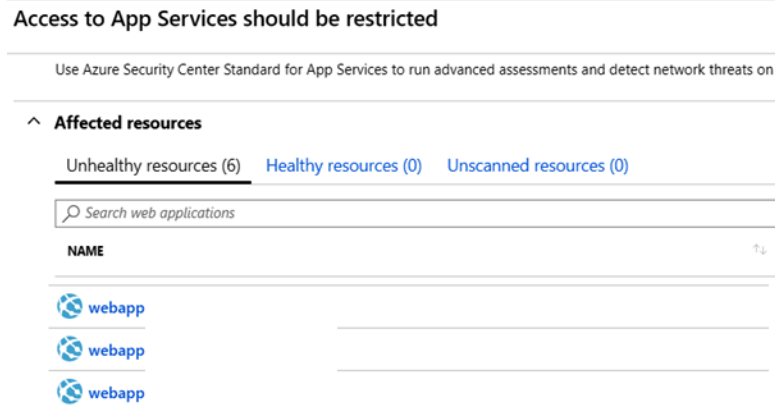


Figure 7.15 – Resource-level security issues

In our example, the warning is **Access to App Services should be restricted**, and non-compliant web apps (obfuscated here) are listed, which lets you drill down into the problem. The warning shown in *Figure 7.15* is raised because the web apps do not have a network restriction mechanism. ASC lets you track down these kinds of issues very easily. Enabling Azure Defender is the best way to start a security audit.

On top of ASC, you can also look at **Azure Sentinel**, a native **Security Information and Event Management (SIEM)** service, which is multi-cloud and usable on-premises. If you do not already have a SIEM, such as QRadar, it is definitely worthwhile to monitor your environment with Azure Sentinel. The major difference between ASC and Azure Sentinel is the fact that ASC inspects your configuration, while Azure Sentinel performs a real-time attack detection. Note that ASC and Azure Sentinel have a few overlapping features, but they can be used together for better security. Azure Sentinel allows you to leverage machine learning to prevent noise and to build a security monitoring platform that is tailor-made to your environment.

As we mentioned in the introduction chapter, cloud-native security means that you embed security controls and practices right into your CI/CD platform. Azure DevOps services and servers (the on-premises version) can be used together with static code and open source library scanners.

JFrog is a third-party product that has a robust offering in that matter, as well as in-container image scanning. **AzSK (Secure DevOps Kit for Azure)** is a free framework that Microsoft uses internally to enforce some security policies against the configuration of Azure services. AzSK's primary strength is to focus on PaaS and FaaS, while ASC initially focused on IaaS. Since ASC also covers more PaaS and FaaS workloads, the interest in using AzSK is reduced. However, it can still be handy to integrate the AzSK extension, which is available for free on the Azure DevOps marketplace (<https://marketplace.visualstudio.com/items?itemName=azsdktm.AzSDK-task>). AzSK has built-in strict policies that can help you find vulnerabilities in ARM templates, right from the CI build, which is very early in the development life cycle.

A very important pillar (which we already covered in previous chapters) is **Azure Policy**, which allows you to monitor (or prevent) deviations. **Azure Arc** even makes it possible to enforce Azure policies across clouds and on-premises, so as to enforce both policies and **role-based access control (RBAC)** wherever your assets are hosted, providing they are hosted on VMs or container platforms.

Of course, proper RBAC management is required to strengthen your security posture. Let's now look at how to set up that identity in the next section.

Zooming in on identity

As stated in the introduction, identity is the primary layer in cloud and cloud-native security. This is even more true when the cloud provider is a *public* cloud provider. We have seen that Azure features a solid collection of network services and features. We also depicted the hub and spoke topology, which is very common and very network-centric. However, in *Chapter 4, Infrastructure Deployment*, we explained the **Azure Resource Manager (ARM)** endpoint. This endpoint is fully public and *cannot be isolated from the internet*, which means you are never completely isolated from the internet. *Figure 7.16* illustrates this:

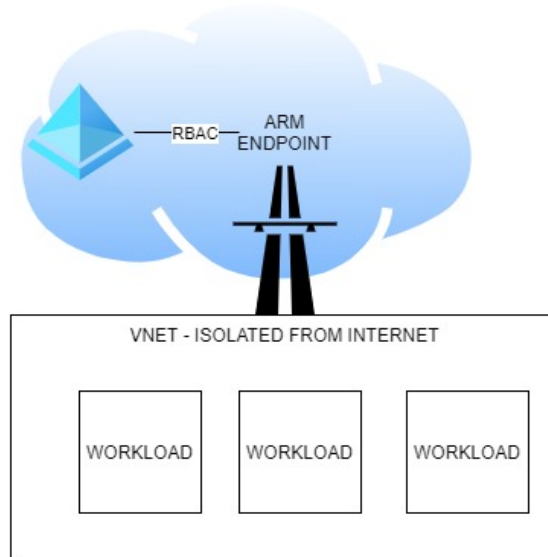


Figure 7.16 – Internet exposure of the ARM endpoint

We see that AAD itself and the ARM endpoint are both 100% internet-facing. AAD enforces RBAC to allow or deny an operation over your Azure infrastructure. Every operation performed by an administrator (using the Azure portal) will cause the portal to call the ARM endpoint on behalf of the logged-in user. You may enforce **Multi-Factor Authentication (MFA)** and even advanced conditional access policies that would prohibit the usage of the portal from outside the company perimeter. That's all fine, but all of this fully remains an integral part of the identity realm.

Even better, conditional access policies, which can trigger MFA or even deny an operation, do not apply to the so-called **Client Credentials Grant**, an OAuth2 flow used in unattended mode. This concretely means that if a service principal has too broad permissions, malicious use of it could totally defeat your network layer. By the way, this is not specific to Azure, because the same applies to AWS CloudFormation. The message we want to convey here is that stacking network layers is not the best security approach in the cloud. It may give you a false impression of security. You're better off focusing on identity, making sure to adopt a **Least-Privilege Approach (LPA)**, as well as to rotate credentials and closely monitor the activity logs. You must switch from network-in-depth to identity-in-depth.

Let's now explore some of the identity features that are available in Azure (see *Figure 7.17*):

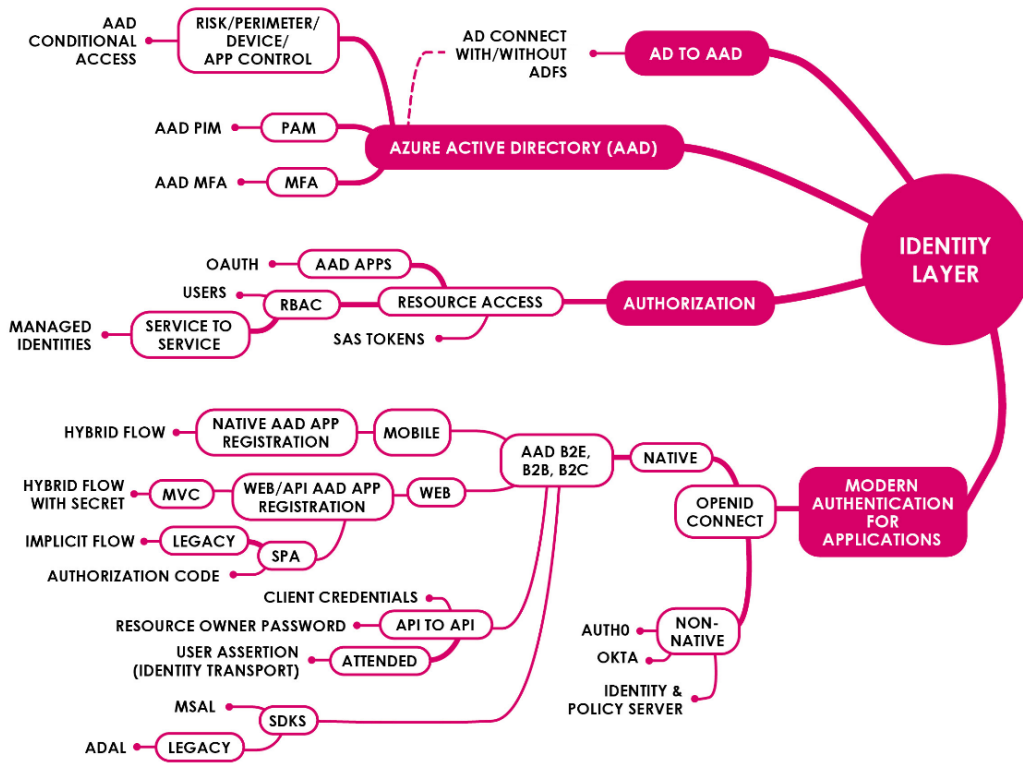


Figure 7.17 – Zooming in on identity features

Needless to say, AAD is the cornerstone of identity in Azure. AAD enables hybrid identities through **Azure AD Connect**, an on-premises solution that is used to synchronize Active Directory identities with AAD, as well as to deploy **Active Directory Federation Services (ADFS)**. ADFS lets you establish a federation between your premises and AAD (among others). When users log in, they are redirected to the ADFS login page (in your perimeter) and are prompted for their credentials, which are validated against your on-premises Active Directory. This makes your ADFS a single point of failure, because in such a setup, user passwords are not synced with AAD. Thus, credential validation can only be performed against your on-premises directory.

Azure Active Directory Pass-through Authentication is an alternative that consists of validating user credentials on-premises. It also validates credentials online, should your on-premises login page not be available. From a pure authentication perspective, it is a more robust approach than ADFS, but it requires user passwords to be synced with AAD, which is often still considered unwise by many organizations. You can also go full cloud and only use AAD.

Conditional Access is available in AAD P1 and P2. It allows you to set up conditional access policies to prohibit a specific activity, as well as to trigger MFA (according to rules that you define). It is a very powerful engine. We briefly introduced it at the beginning of this section, when explaining why identity remains the most important layer in the cloud. You may target conditional access policies toward specific users or groups, or to specific apps.

In AAD P2, you can leverage AAD **Privileged Identity Management (PIM)** to only elevate user privileges (RBAC) when required. This is completely in line with the least-privilege principle. AAD PIM keeps track of elevation requests and only grants higher privileges for a certain duration. Users must justify the reason why they need more privileges and approvals can be enforced.

Regarding resource access, we already mentioned SAS tokens and managed identities, which we will explore in depth later in this chapter.

When it comes to modern authentication, we often refer to **OpenID Connect (OIDC)**. OIDC is an identity layer on top of OAuth2, which is much older. Both OAuth2 and OIDC are pure internet-based protocols, hence the reason why they are typically not very well-known by on-premises security architects. AAD and AAD B2C are both native services that do fully support OIDC. Identity Server, Auth0, and Okta are typical alternatives, although AAD will always remain in the picture, no matter what setup you have.

To leverage AAD and AAD B2C endpoints, as well as any Microsoft identity, developers should use the **Microsoft Authentication Library (MSAL)**. You also may still occasionally see the **Active Directory Authentication Library (ADAL)**, which has now become legacy, but it is still heavily used today. In OIDC, it is important to understand the different authentication and authorization flows, and more importantly, unattended flows, which can represent a risk to your environment. All user-based flows are subject to conditional access policies (if any). You have a ton of means to control your end user identities. The following are two unattended flows:

- The **Resource Owner Password Credentials** grant: We could compare this grant with service accounts or technical accounts that we use on-premises. For example, you can have an application pool identity on IIS. It is like a user identity, with a username and a password, but is not used by end users. This grant is also subject to conditional access policies.
- The **Client Credentials** grant: This grant type is used for service-to-service communication, and it is not at all related to user identity. The credentials can be a pair of client ID/client secret or client ID/certificate. This grant is not subject to conditional access policies.

The Client Credentials grant is by far the most dangerous one, but it is also inevitable, so you can't simply forbid it. There are countless scenarios that require this grant, including integrating with QRadar, Splunk, and Dynatrace, to name a few. However, if a malicious insider has access to a pair of credentials, they can make use of them from anywhere in the world and from any device. They will be able to request an access token to AAD and talk to the ARM endpoint (because it is public, independent of your network plumbing) to manipulate your environment. That is why it is key for you to have a frequent rotation of such credentials, as well as adopting the least-privilege principle.

You should also monitor the usage of such identities more closely. Managed identities also leverage the Client Credentials grant, but the good news is that the so-called credentials cannot be leaked, so you should use them as much as possible.

On that topic, let's now explore managed identities in depth.

Delving into the most recurrent Azure security topics

In this section, we will focus on the most recurrent security features, which are highly discussed and that you will surely be confronted with. They are also not typical of the traditional on-premises security arsenal, which often makes security experts clueless on the matter. After reading this section, you will be more familiar and more confident in any upcoming security conversations. Let's start with **Azure managed identities**.

Exploring Azure managed identities in depth

Azure managed identities solve a problem that was around for ages: storing credentials. We know that we can use Azure Key Vault to store credentials, but we also know that you need another pair of credentials to access the credentials stored in Key Vault.

Where do you store them? We have a chicken and egg problem. That is exactly what managed identities solve. With managed identities, Azure will automatically generate a pair of credentials and make them available to the execution environment when requested. There is no need to manage or store these credentials anymore. The application will receive them on request. Moreover, these credentials can only be used from within the application itself, through an internal endpoint that cannot be accessed outside of the application. *Figure 7.18* shows how managed identities work:

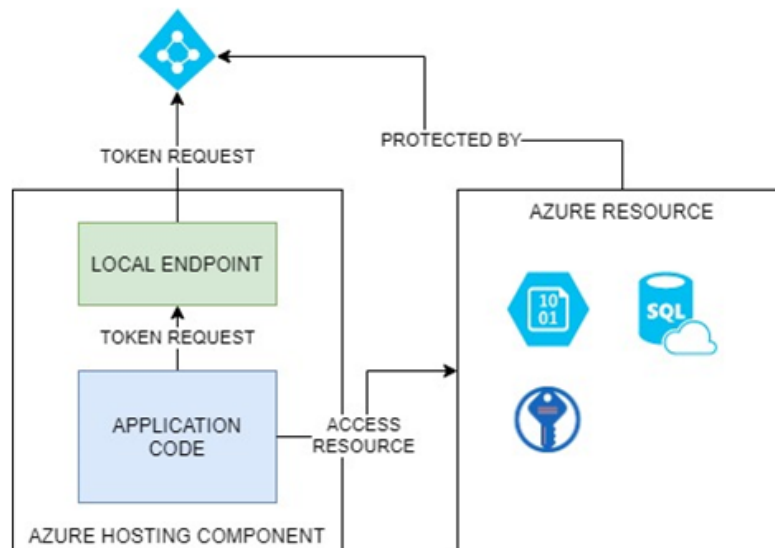


Figure 7.18 – Managed identities

For example, if you have two different Azure Functions apps, with **Managed System Identity (MSI)** enabled, they will each have the following environment variables defined:

- **MSI_ENDPOINT**: This could be `http://127.0.0.1:41772/MSI/token/` for one function app and `http://127.0.0.1:41773/MSI/token/` for the other.
- **MSI_SECRET**: The secret generated by Azure for this service instance.
- **IDENTITY_ENDPOINT and IDENTITY_HEADER**: The same as the preceding variable, but it has different names because of a change in the way a token can be requested.

The following non-exhaustive list is of Azure hosting services that can be MSI-enabled:

- An app service
- A function app
- A VM
- A container running in AKS through AAD pod identities
- A container instance
- A data factory pipeline

The aforementioned list is not exhaustive, and it keeps evolving over time. Similarly, the number of Azure resources (that support AAD authentication) increases over time. Managed identities are assigned a role against the target resource, through Azure RBAC. However, not every Azure resource supports the managed identity model (for example, Azure Cosmos DB). In such a situation, you have the following two options:

- Store the Cosmos DB access key into Azure Key Vault, enable managed identities for the client app, and grant the managed identity access to Key Vault. With this setup, the client app is able to pull the access key from the vault and can authenticate against Cosmos DB.
- Enable managed identities for the client app and grant it an RBAC role over the target resource. In our example, the *Cosmos DB Account Reader* role could be granted, to let the app retrieve the access key through the ARM endpoint (instead of pulling it from Key Vault).

Whatever solution you choose, you do not have to store credentials anywhere. As a best practice, managed identities should always be preferred over storing credentials or using another authentication mechanism. Note that managed identities come in two flavors: **user-assigned** and **system-assigned**.

With the system-assigned approach, the identity is automatically created and assigned by Azure to a given service. It means that you have a 1:1 mapping. Each service has its own identity. With the user-assigned approach, the identity object is a separate resource that can be assigned to one or more services. In both cases, you can manage these identities as if they were users or groups. You can add them to existing groups and grant the group access to the target resources.

Adding managed identities to groups allows you to prevent ad hoc role assignments. On the other hand, there is no out-of-the-box ARM template support to automate this process for system-assigned identities. This can be done through Azure CLI or PowerShell, but it requires permission to manage users and groups. This is where user-assigned identities can bring you the following benefits:

- You can pre-create a user-assigned identity as a separate object, and then add it to a group that has been granted the necessary permissions.
- You can use ARM templates to associate the user-assigned identity to a resource.
- In some very dynamic scenarios, such as having dynamic ACIs (or containers within AKS), user-assigned identities are a better fit. They are already granted access to the target resource(s), prior to the launch of the application code. With system-assigned identities, you should still grant access or add it to a group once the resource is provisioned. This is not ideal for extremely dynamic workloads.

Important note

Even if this may seem obvious, we want to draw your attention to the fact that managed identities are only for Azure-to-Azure authentication. They cannot be used from applications running outside of Azure. Another aspect to keep in mind is that managed identities are associated with a tenant (AAD). Moving a subscription with resources that have managed identities to another directory will not create these identity objects in the target tenant.

Let's now have a look at another recurrent way of authenticating against a target Azure resource.

Demystifying SAS

SASes have been in use in Azure for quite a long time, although they tend to be gradually replaced by AAD OAuth tokens. They are still typically used with Azure Service Bus, Azure Storage, and Azure Event Hubs (to name a few). SAS is **Hash-Based Message Authentication Code (HMAC)-SHA256** that contains the encrypted hashed value of the query string parameters. HMAC is often used with digital signatures. To illustrate this with a concrete example, let's explore the following signature, which is an example of authentication against Azure Service Bus, to create a new subscription:

```
Authorization: SharedAccessSignature
sr=https%3A%2F%2Fthemapbook.servicebus.windows.net%2Fdatasets%
2FSubscriptions%2Faci-large%2F&sig=hc1TVmzT%2BQrojnQN6vjc9CtL
3WcPKVU01QqdABUVjY%3D&se=1608650456&skn=key
```

We see that the SAS token is transmitted over the Authorization HTTP request header. In this case, it has the following four parameters:

- `sr`: The service resource endpoint that is targeted by the request. In this case, we target `https://themapbook.servicebus.windows.net/datasets/Subscriptions/`.
- `se`: This is the epoch expiration time of the SAS.
- `sig`: This is the signature, meaning that it's the Base64-encoded hash value of both `sr` and `se`, combined and separated by a line feed.
- `skn`: The name of the symmetric key that was used to compute the HMAC.

The `sig` parameter is received by the resource server, which recomputes it given the `sr` and `se` values it received in clear text. The resource server compares it with the recalculated value, and if they match, the server authorizes the request. If not, it rejects the request with a 401 return code.

The SAS computation process involves both hashing and encryption. This makes it robust against a man-in-the-middle attack, because the symmetric key is not transmitted over the wire. Tampering with the request would inevitably result in another hash value on the resource side, which would lead to a hash mismatch. Thus, the request would be denied by the resource server. So, the SAS mechanism is pretty robust. However, keep in mind the following aspects:

- The symmetric keys that are used to encrypt the hash value must be kept secret and should not be leaked.

- Anyone that has a pointer with a SAS key will be able to access the resource. This is also true with OAuth2 access tokens. If you find the keys to a house and you know its location, you can enter even if you're not the owner.

With the aforementioned points in mind, you should apply the following principles:

- Plan for frequent key rotations.
- Plan for short SAS lifetimes. Often, it is possible to generate SAS that only has a lifetime of a few minutes. If that is enough time to perform the operation, you should not make it last longer. A leaked expired SAS is not a risk anymore.
- Plan for an LPA. Capabilities here depend on the resource server. For example, with Azure Service Bus, you can create custom shared access policies that let you define keys for read-only or write-only. With Azure Storage, you have much more granularity that you can use to your advantage, to implement your LPA.

People often wonder why Azure services that support SAS authentication have two keys: primary and secondary. This is to accommodate key rotation. To rotate keys without disrupting clients, you should follow this sequence:

1. Copy the primary key value into the secondary key.
2. Regenerate a new primary key. While you are doing this, your clients can keep using their old primary key.
3. Update clients to use the new primary key.
4. Rotate the secondary key to invalidate the old primary key.

Executing the preceding steps in this sequence will ensure a smooth key rotation.

Understanding APL and its impact on network flows

APL and **Azure Private Endpoint (APE)** are closely related and provide long-awaited capabilities. The purpose of APL and APE is to remove public endpoints from well-known PaaS storage services, such as Azure Storage and Azure SQL. However, they can also isolate hosting services (such as Azure App Service) from the internet. When APE is created in a subnet of your choice, a system route of /32 is created to APL, which proxies the target resource. *Figure 7.19* shows how APL and APE work for an app service:

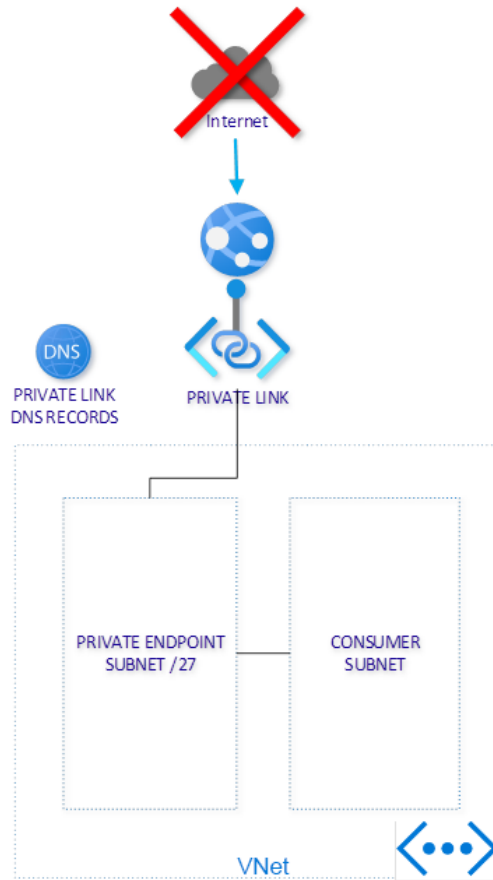


Figure 7.19 – APL and APE

By design, an app service is publicly accessible over the internet. With APL, you can hide it behind a private IP address, for which you designate the target subnet. Every consumer must be in a network perimeter that can reach out to that IP address. A private DNS zone hosting the private DNS records must also be associated with each consumer VNet, to resolve the private link endpoint.

In *Figure 7.19*, we connect to APE from a subnet that is in the same VNet. You can of course connect from anywhere, providing connectivity is established across VNets (remember the hub and spoke). We clearly see that the private link-enabled resource, in this case, the app service, is not itself inside the VNet. This means that for the time being, only inbound traffic is controlled, not outbound. If our application is hosted on an app service or an Azure function, the outbound traffic will still use Azure system routes.

So, keep in mind that enabling APL for a service does not have any impact on its outbound traffic. This can be mitigated by implementing App Service VNet integration on top of APL, or by running the app service in an ASE. Moreover, APE comes with the following limitations:

- NSGs do not apply to private endpoints, which clearly means that associating an NSG with a subnet that contains APE will have no effect.
- User-defined routes do not apply to private endpoints. This is the same as the preceding point.

This means that from a target perspective, there is not much you can control. The inbound traffic will skip both NSGs and UDRs, meaning that it won't be routed to the hub. It is, therefore, an exception to the way a typical hub and spoke topology works. Security people assume that both inbound and outbound flows will all be tracked by the NVA or Azure Firewall. So, from a target perspective, this assumption is wrong. Basically, you must start from the consumers and give APE and APL specific care. There are a number of mitigation scenarios described by Microsoft at <https://docs.microsoft.com/azure/private-link/inspect-traffic-with-azure-firewall> that let you better define how to control APE and APL.

APL also comes with numerous trade-offs and/or limitations (as of December 2020). The following are a few example limitations:

- When enabling a private link for Azure Container Registry, you cannot perform image vulnerability scanning with ASC anymore.
- When a private link is enabled for a resource, other Azure services that are not VNet-integrated or self-hosted cannot connect to that resource anymore, except for the so-called *trusted services*. The problem is that not every resource supports these trusted services. For example, Azure Storage's firewall can whitelist trusted services, but not Azure SQL or Azure Cosmos DB. This may lead to quite complex and convoluted architectures.

As a precautionary measure, you should always double-check the limitations/drawbacks that may apply when using a private link against a particular resource, and the potential impact on the other components of the solution. Also, something to keep in mind with a private link is that it does not really remove the public endpoints; it just changes the behavior of their firewall. This also varies by service. Here are a few examples of such variations:

- A private link applied to App Service and Azure Functions denies all traffic against the public endpoint by default.

- A private link applied to a Storage account leaves the public endpoint available. It is up to the cloud consumer to deny all public traffic (if desired).
- A private link applied to Event Hubs sets the firewall to accept only traffic from *selected networks*, which means *everything* by default.

You should not assume that enabling a private link will automatically deny all public traffic; it is just an opportunity to deny it all. This leads us to our next section, the resource firewalls.

Understanding Azure resource firewalls

Resource firewalls are another very common topic with regards to Azure. By resource firewalls, we mean the Azure Storage firewall, Azure SQL firewall, Azure Cosmos DB firewall, Azure Container Registry firewall, and so on. At the time of writing, APL and resource firewalls are mutually exclusive. When a private link is enabled for a resource, every actor that can connect will be authorized by the resource. Of course, you may block the traffic in Azure Firewall or your NVA, but not at the level of the resource itself. Azure resource firewalls apply only to the public endpoints of the PaaS resources. Their feature set may vary from one service to another, but in general, they let you do the following:

- Whitelist individual IP addresses or CIDR.
- Whitelist VNet subnets through SEs.
- Define exceptions, which often differ from one service to another. Azure Storage is the most comprehensive firewall from that perspective, but it is also very permissive by default, because no firewall rule is enforced by default. Some exceptions might also invalidate the other rules. For example, in Azure SQL, you have the possibility to allow *Azure services and resources to access the server*, which somehow invalidates a more restrictive rule. You must pay attention to the exceptions that you make.

SEs are the ancestors of APL. The key difference between SEs and a private link is that with SEs, the traffic still leaves the VNet to hit the public IP of the PaaS resource. With private links, the traffic remains 100% internal to the VNet, and you can leverage private links with on-premises consumers, unlike with SEs. One of the only benefits of SEs is that you do not have to do anything DNS-wise, because they use the out-of-the-box public Azure DNS, while private links require the use of an extra Azure Private DNS zone.

To be cloud-native, firewall rules should be deployed through CI/CD declaratively or imperatively and enforced through Azure Policy to prevent or control deviations.

Let's now review our Contoso use case and add some security bits to it.

Adding the security bits to our Contoso use case

In this section, we will review our Contoso use case that we started in *Chapter 2, Solution Architecture*, and improved in *Chapter 5, Application Architecture*. However, none of our diagrams included security-specific portions. It is time to fix this and see where to use some of the features that we have explained throughout this chapter. *Figure 7.20* illustrates what we ended up with in *Chapter 5, Application Architecture*:

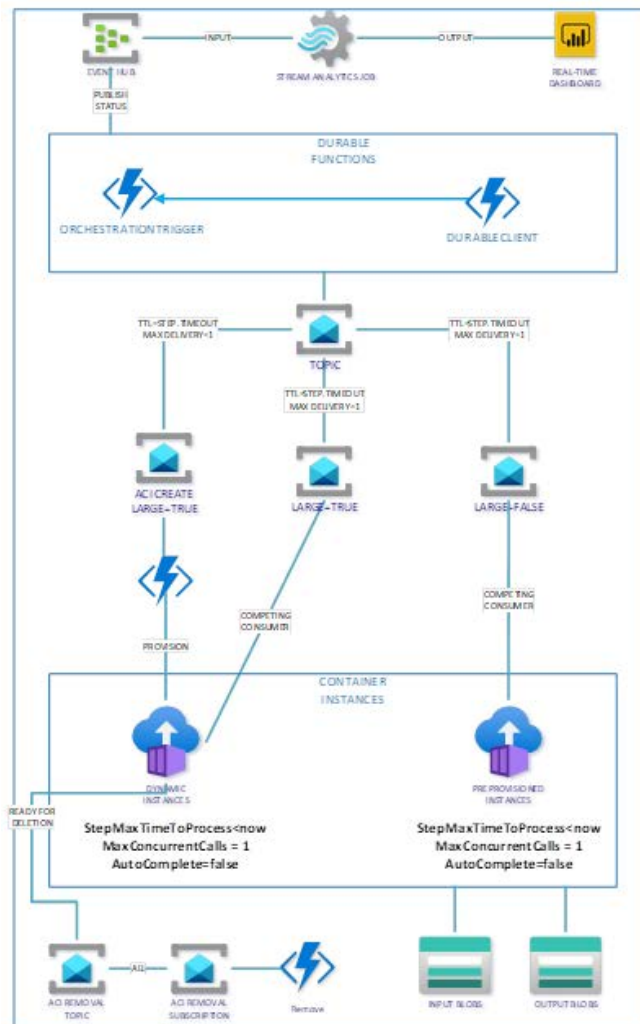


Figure 7.20 – Reminder of the Contoso use case from Chapter 5

As you can see, there is nothing specific about security. For the sake of simplicity and brevity, we will get rid of the Power BI and Stream Analytics services. So, our new functional flow is now as shown in *Figure 7.21*:

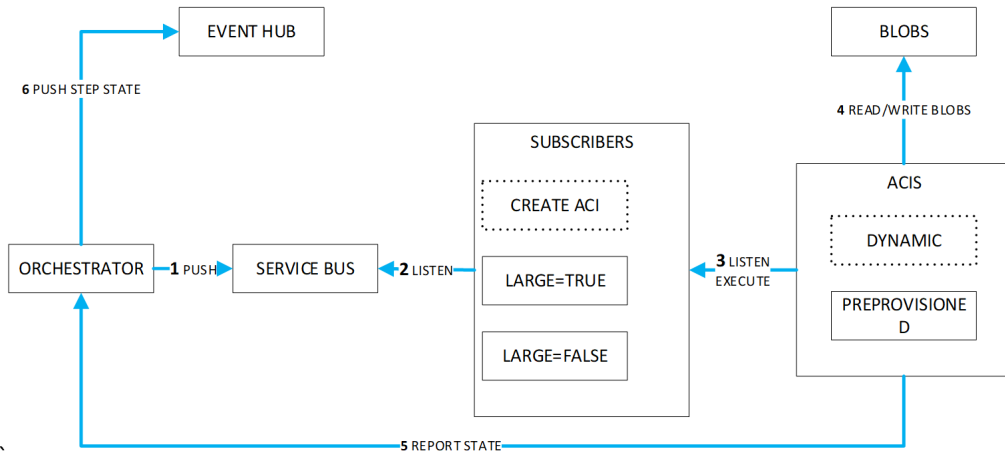


Figure 7.21 – Revisited flow without Stream Analytics and Power BI

For the security bits, we are interested in the interactions between the components, to understand how we can apply a least-privilege principle for this solution. We see the following:

- The orchestrator interacts with both the service bus and the event hub.
- Our subscribers interact with the service bus, and some of them must be able to provision ACIs dynamically. So, they also need some write permissions.
- Our ACIs, whether dynamic or pre-provisioned, must be able to read messages from the service bus, and then report their state via a callback URL. They also read input blobs and write output blobs. These blobs represent our data.

A serverless-only version of this solution is shown in *Figure 7.22*:

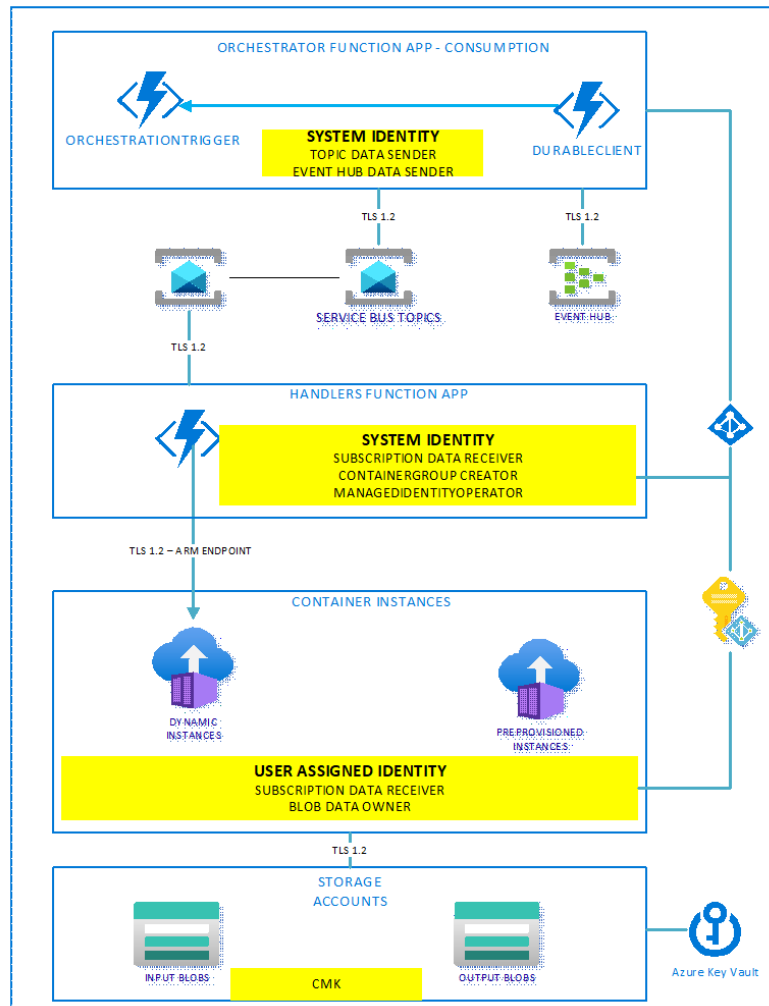


Figure 7.22 – Serverless-only solution

All our interactions run over the public internet. Remember that serverless usually implies that you have no control over the execution environment, meaning there is no (or little) possibility to apply firewall rules at a data level, and there is also no possibility to reach out to private endpoints. The security portion of *Figure 7.22* mostly relies on identity and encryption. Let's see it step by step, as follows:

1. Resource access is entirely ensured by managed identities. We use both system and user identities.

2. Our orchestration block has its own function app, with a system identity that is granted the topic data sender and event hub data sender. The orchestrator must send messages to both the service bus and event hub.
3. Our handler's function app is a separate app, for better RBAC segregation. Its system identity is granted the subscription data receiver (service bus). It also needs to provision ACIs on the fly. For this, we will create a custom RBAC role that allows just this. At the time of writing, such a role does not exist. It will attach a user identity to the ACIs, which requires the managed entity operator role against the target identity.
4. Our dynamically provisioned ACIs will make use of an existing user identity. This identity is granted the roles of subscription data receiver and blob data owner, because it needs to read and write blobs. The ACIs do not expose any inbound port, but they need to reach out to Azure Storage. We use a user identity instead of a system identity, because we do not want to create too many identities for these ephemeral instances. Also, the necessary permissions are already granted to the identity.
5. Encryption in transit happens over TLS 1.2 for all communications.
6. Encryption at rest is ensured with a CMK, which is stored in the vault.

The pros of this solution are as follows:

- This is the cheapest possible solution. (It could be only a few dollars per month.) It is also the easiest and fastest, in terms of deployment.
- All credentials are entirely managed and rotated by Azure itself. Moreover, these credentials can only be used from the services themselves. The only residual risk is the code itself, which could be vulnerable. This could be addressed with a pen test.
- Roles are reduced to the strict minimum required, so our least-privilege principle is well respected.
- Encryption in transit and at rest are both ensured.

The cons of this solution are as follows:

- Our storage account is internet-facing.
- Our orchestrator (HTTP trigger) is also internet-facing.

We could mitigate the second item by putting the orchestrator behind an API gateway, and then we could enforce a few extra policies and/or a WAF. For the data aspect, it is trickier. Having a public storage account might be acceptable, depending on the data classification. We may also wonder what the exact risk is that we are taking. A storage account is already a hardened service, with only port 443 opened (because we can close 80). It has an Azure API around it with an intended use.

Brute forcing our identity piece is useless because we know that managed identities can only be used from within the services themselves. We do not make use of the storage keys, and we can monitor who/what attempts to list the keys (and rotate them if required). Running DoS/DDoS attacks against a storage account is rather something that Microsoft must deal with. We use our own DEK, which reduces the data exploitation (should the data be leaked). So, this might just be good enough. Here, we had a network-free solution.

However, to be comprehensive and because we know the network still remains the primary layer for most security folks, *Figure 7.23* shows an alternative, with everything fully privatized. Of course, as we explained before, costs and complexity rise:

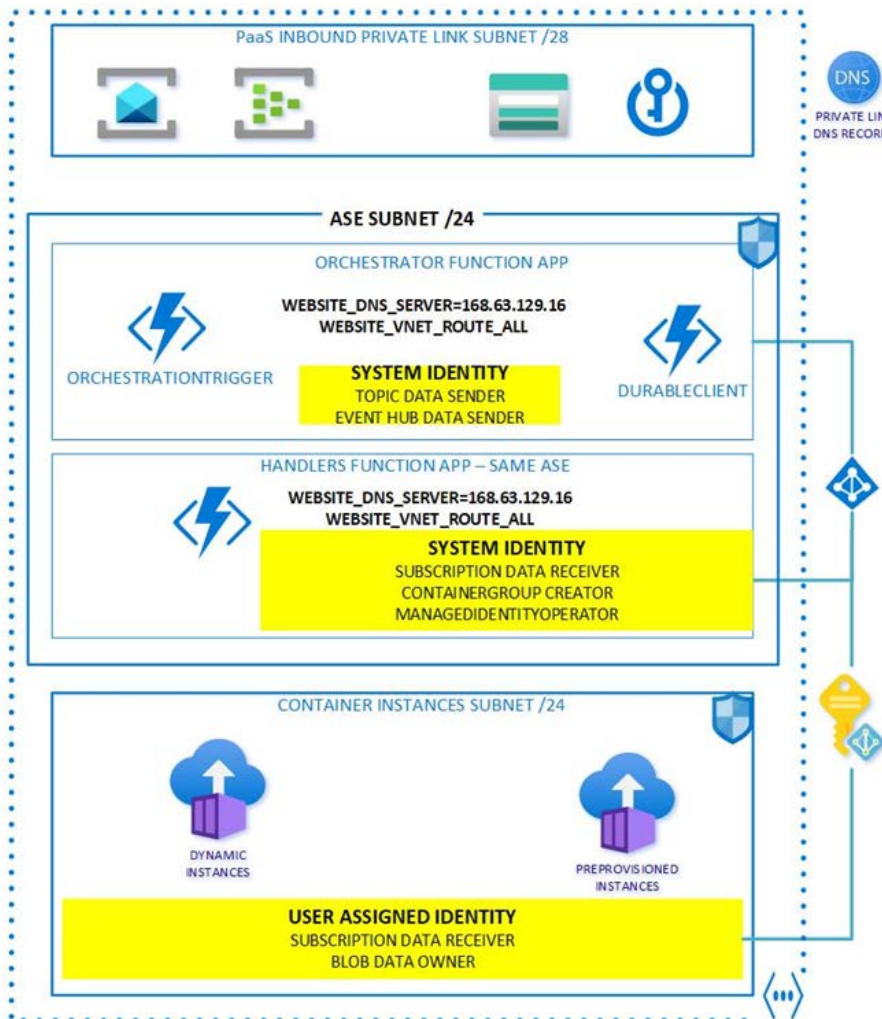


Figure 7.23 – Network-centric approach

In this architecture, we isolate everything from the internet. The identity and encryption pieces remain unchanged. The following aspects are different:

- All our PaaS data services are private link-enabled, meaning that the inbound connectivity is fully private. We have a dedicated small subnet for all our private endpoints. Note that Azure Storage still allows public connectivity, but it denies everything by default.
- We run our function apps on an ILB ASE. In the app settings, we point to the specific DNS endpoint that allows function apps to find private link-enabled services. We take a /24 subnet size, as recommended by Microsoft.
- Our container instances sit in a separate subnet. Here again, we take a /24 subnet size.

The pros of this solution are as follows:

- We have no more public-facing endpoints.
- We keep a good identity layer.
- Combining both layers makes it stronger because we reduce the number of attack vectors.
- This fits with the hub and spoke topology, although we considered it as a standalone solution in this diagram.
- We have dedicated compute with the isolated app service plan, preventing the noisy neighbor effect of serverless compute.

The cons of this solution are as follows:

- Costs rise with the ILB ASE and the isolated App Service plan, even if we share the plan across function apps. With the current Azure offering, only for the ILB ASE and the isolated App Service plan, we can count around \$1,800/month (for a single instance of the plan). In the future, the \$1,000 monthly flat fee should be gone. But this is still way more expensive than the serverless solution.
- It is more IaC work, since we need to combine a private link, ASE, and all the DNS plumbing.
- Unlike the previous solution (*Figure 7.22*), this one is not elastic anymore, because our subnet sizes will define our capacity. Indeed, concurrent container instances will need concurrent IP addresses to be available. The same goes for function app instances. For the serverless solution, we did not have to worry about this because Microsoft allocates/deallocates resources per the actual consumption needs. In the current solution, we could be tempted to overscale to avoid being blocked, but we may waste IP addresses as well.

- Should our solution evolve over time, our network-centric approach might prevent us from bringing extra features or services, or force us to use premium-only services, which will dramatically increase costs. For example, should you require Azure APIM, only the premium tier would be possible. Front Door would also not be usable in such a setup.

As you can see, the privatization of services often adds more complexity and incurs higher costs. The message we want to convey here is that you should formulate a real risk assessment, and you should not impose a one-size-fits-all model, because sooner or later it becomes an impediment. Keep in mind that PaaS and FaaS services are not mere VMs for which you are entirely responsible (in terms of security). A cloud-native approach consists of improving your identity and encryption layers, as well as having an improved security incident prevention-detection resolution. Let's now recap the chapter.

Summary

In this chapter, we introduced the vast security landscape of Azure, which deserves an entire dedicated book. We gave you a glimpse into cloud-native security, and what it implies in terms of mindset and technology choices.

We explained to you why identity is the primary layer of defense in the *public* cloud, and we highlighted a few trade-offs that are incurred by a network-centric approach. A network approach is often the default approach, which is inspired by decades of traditional security practices on-premises. We saw that Azure has quite a lot of built-in security features and services that we can use to our advantage not only to secure our Azure workloads but also to secure other clouds and even on-premises systems.

Lastly, we reviewed our initial Contoso use case, from the eyes of a security architect, by adding two specific security views to our diagram. By now, you should be better equipped to tackle Azure-specific security topics as well as to deal with cloud-native applications that carry their own way of envisioning security.

The next chapter is a recap of what we have learned up to now, as well as an introduction to some typical industry-specific architectural scenarios.

Section 3: Summary

Finally, we will go back through the architecture of the book, showing many use cases, per industry, that expand on how the architectural perspectives come together and result in a cohesive solution in Microsoft Azure.

In this section, the following topic will be covered:

- *Chapter 8, Summary and Industry Scenarios*

8

Summary and Industry Scenarios

In this chapter, we will go back through the architectures covered in the book, showing various example scenarios per industry that expand on how the architectural perspectives come together to result in a cohesive solution on Microsoft Azure. The deeper you dive into the industry architectures that most closely resemble your projects, the wiser and more prepared and impactful you become.

This chapter covers the following topics:

- Revisiting our architectures
- Automotive and transportation scenarios
- Banking and financial services scenarios
- Gaming scenarios
- Healthcare scenarios
- Manufacturing scenarios
- Oil and gas scenarios
- Retail scenarios
- The unique values of this book

Revisiting our architectures

First, we'll revisit our architectures by summarizing each chapter of the book. This allows the opportunity to refresh your memory and offer a key appeal from each chapter, one important takeaway that could greatly impact your cloud architectures.

Sample architecture

In *Chapter 1, Getting Started as an Azure Architect*, we explained the basic idea behind the maps in this book, why we're using them, and how you can use them to further explore architectures. We provided a sample map (see *Figure 8.1*) to properly explain how the maps are structured:

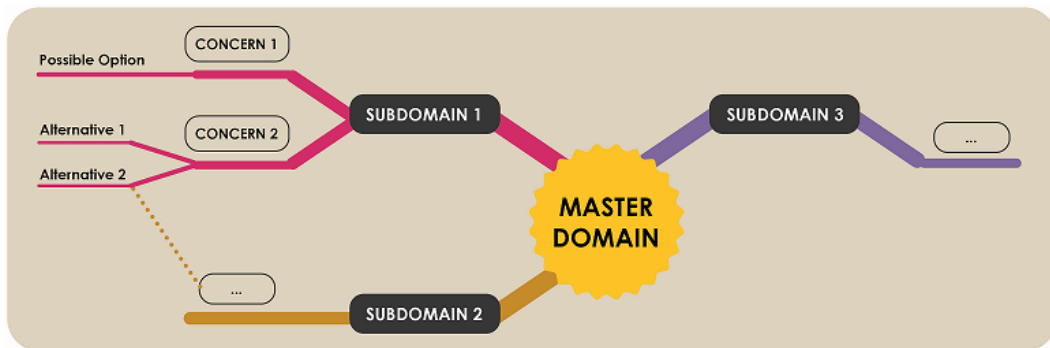


Figure 8.1 – Our sample map

The sample map demonstrates the **subdomains**, **concerns**, **options**, and **alternatives**. Our larger-level architecture map focuses on the master domain and subdomains. When we zoomed in, we showed you a fuller map that includes the concerns under a given subdomain. Next, we browsed the different types of architects and showed you what makes them unique, different, and valuable.

Then we covered the essential cloud vocabulary, discussing the jargon that is used in this book. (We used a lot of acronyms with *aaS* at the end of them.) Then, we discussed what makes a cloud journey successful, showing the full product life cycle of a cloud solution and deployment. All of our maps are available as VSD and JPG files on GitHub.

Appeal

As many of our maps would be too large for this book (at their full size), we ask that you find the VSD and PNG files of the maps on GitHub: <https://github.com/PacktPublishing/The-Azure-Cloud-Native-Architecture-Mapbook>. Open the chapter folder, and then open the Diagrams and/or Maps folders to find them. Exploring the larger map files allows you to see all the details and to revisit the maps on your architecture journey.

Solution architecture

In *Chapter 2, Solution Architecture*, we dug into our first architecture! We took you on a long (but hopefully enjoyable) journey through the eyes of a solution architect. See the architecture in *Figure 8.2*:



Figure 8.2 – The solution architecture map

In GitHub, we provided the full architectures as Visio files and PNG images. We then zoomed in on **Systems of Engagement (SoE)**, **Systems of Record (SoR)**, **Systems of Insight (SoI)**, and **Systems of Integration (SoX)**.

We covered **Event-Driven Architectures (EDAs)**, cross-cutting concerns, monitoring, **Continuous Integration and Continuous Deployment (CI/CD)**, identity, connectivity, governance/compliance, and containerization. In *Chapter 2, Solution Architecture*, our zoom-in on **Identity** looked a little like a baby octopus, but with only four tentacles (also more orange than the average octopus, if you have the fortune of seeing it in color). See *Figure 8.3*:

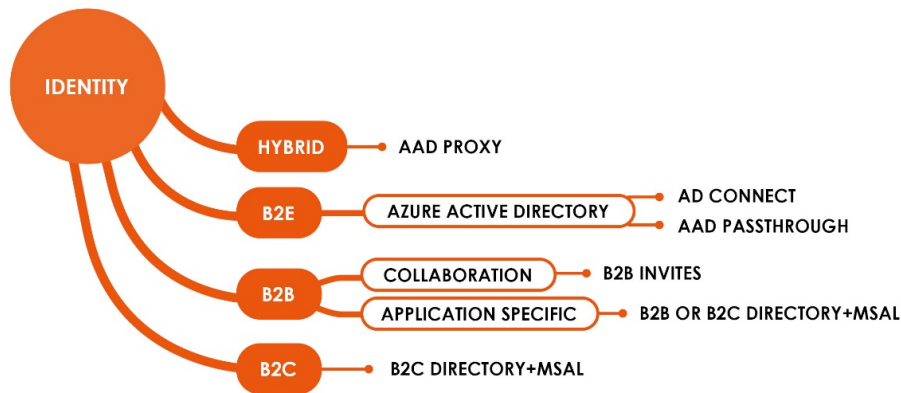


Figure 8.3 – Zoom in on identity (a baby octopus)

That's when we hit you with a massive solution architecture use case, complete with a business scenario (Contoso needed yet another configurable workflow tool), reference architecture, scripts we used, and the full solution (available in GitHub). We also walked you through the orchestration process. This architecture book got practical fast!

Appeal

Take the time to look through the code of our sample and follow along with our instructions. Also, follow along with the details about the orchestration process. You'll learn a lot by going through those steps!

Infrastructure architecture

Chapter 3, *Infrastructure Design*, revealed our infrastructure architecture map (see Figure 8.4):

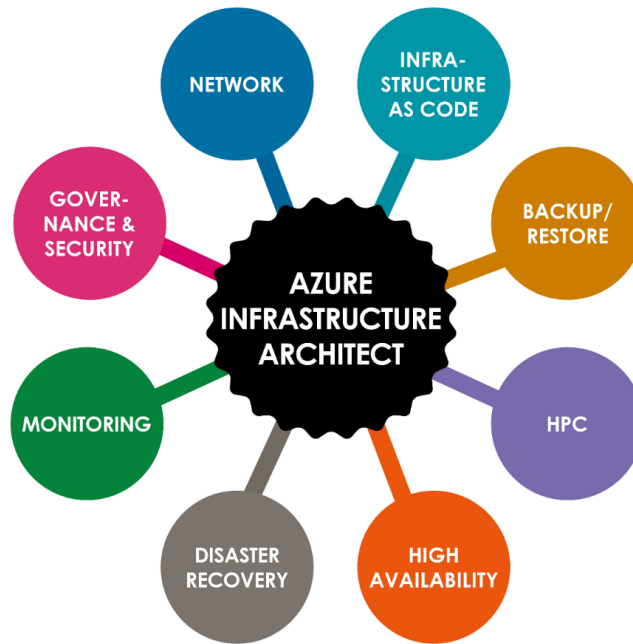


Figure 8.4 – The Azure infrastructure architecture map

We zoomed in on networking, monitoring, high availability, disaster recovery, backup and restore, and **High-Performance Computing (HPC)**. Then, we gave you a bonus architecture with the **Azure Kubernetes Service (AKS)** architecture map, thoroughly diving into the container infrastructure! Wasn't that a pleasant surprise?

Appeal

AKS is not a service like the others. It comes with its own universe! Do not underestimate its impact within your infrastructure architecture. Also, keep in mind that you'll find it very challenging to implement a consistent disaster recovery-compliant architecture with **Functions as a Service (FaaS)** and **Platform as a Service (PaaS)**.

Azure deployment

Next came the chapter you would expect to come after *Chapter 3, Infrastructure Design: Chapter 4, Infrastructure Deployment*. There we discussed CI/CD, looking in depth into the process for the development, build, and release cycles, up to where you deploy on Azure. Next, we presented the Azure deployment map (see *Figure 8.5*), and we zoomed in on scripting (Azure CLI, PowerShell, Cloud Shell), **Azure Resource Manager (ARM)** templates, Azure Bicep, and Terraform:

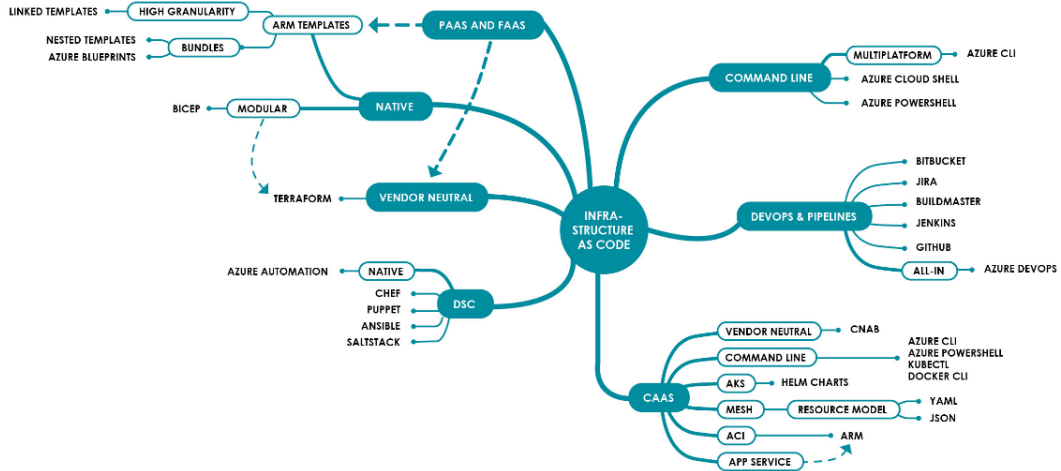


Figure 8.5 – The Azure deployment map

We provided instructions to help you directly get started with your deployments. We then zoomed in on a full reference architecture that leverages Azure DevOps and native ARM templates and includes a full **Infrastructure as Code (IaC)** factory! We cannot stress enough how important the ARM endpoint is; we thoroughly recommend you embrace and perfect the art of working with ARM!

Appeal

Read carefully through our sections on ARM and ARM templates. This is an important way to deploy your application as productively as possible, without reinventing the wheel. Also, make sure you thoroughly consider Azure Bicep and Terraform. These tools could change many of your practices! Finally, we strongly suggest you build out your DevOps processes and tools. If you build out your CI/CD pipelines, you will ensure you can deploy whenever you need to.

Application architecture

In *Chapter 5, Application Architecture*, we were able to unravel some common cloud and cloud-native application patterns. We discussed the pillars of cloud development: DevOps, CI/CD, and PaaS/FaaS. We then contrasted that with cloud-native development, which consists of the following pillars: DevOps, microservices, CI/CD, and containers. We explored the Azure application architecture map (see *Figure 8.6*) to understand how to design a cloud-based application:

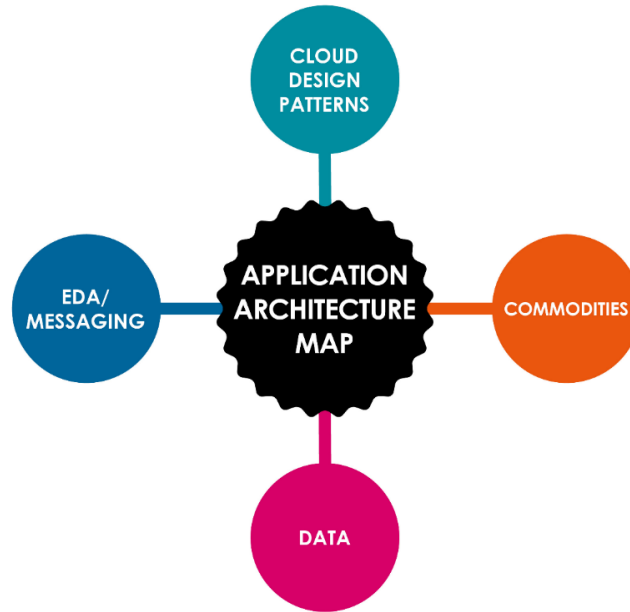


Figure 8.6 – The application architecture map

We zoomed in on data (although we then covered it extensively in *Chapter 6, Data Architecture*) and some key cloud design patterns. We covered **CQRS**, which stands for **command query responsibility segregation**, which segregates the commands (writing) and querying (reading) to increase speed and scale. The Event Sourcing pattern stores the events in an event store and publishes them for consumption.

Our cloud-native design patterns included the Anti-Corruption Layer, Ambassador, Circuit Breaker, Retry, and Cache-Aside patterns. Our API patterns included Gateway Aggregation, Gateway Routing, Gateway Offloading, and Backends for Frontends. We also covered the SAGA pattern (with which you can manage data across your microservices). Next, we covered EDAs, and we drilled into how you would build your own microservices solution.

Appeal

We cannot fully express the importance of the ecosystem in cloud and cloud-native applications. As an application architect, make sure you learn and leverage the Azure and K8s ecosystems. Don't fall into the trap of trying to reinvent the wheel (or deny that the wheel needs to exist).

Data architecture

Chapter 6, *Data Architecture*, showed us five top-level groups (see *Figure 8.7*): big data, modern, traditional, **Artificial Intelligence (AI)**, and other (which comprises our cross-cutting concerns):

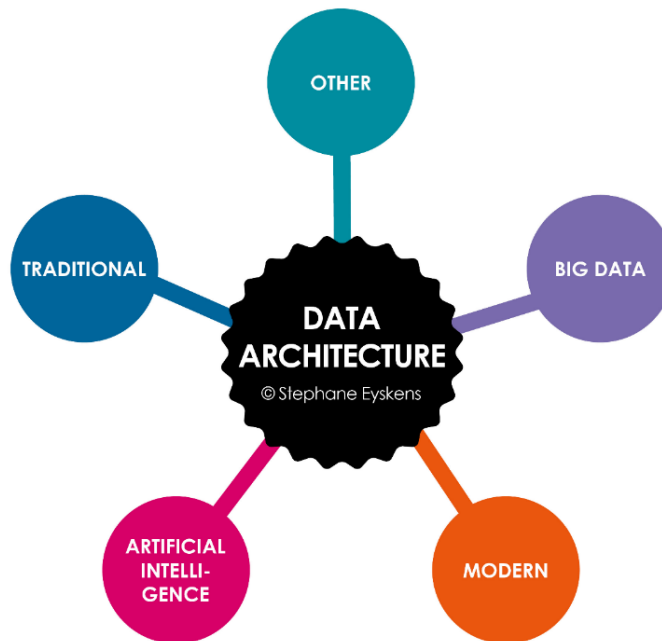


Figure 8.7 – The data architecture map (reduced)

We analyzed the various data practices, and then we looked into the details of **online analytical processing (OLAP)**, **online transaction processing (OLTP)**, **extract, transform, and load (ETL)**, **relational database management system (RDBMS)**, **extract, load, and transform (ELT)**, NoSQL, storing keys/values, and document stores. We then dove into the big data services, including for open source (HDInsight), ingestion (IoT and Event Hubs), and analytics. See *Figure 8.8*:

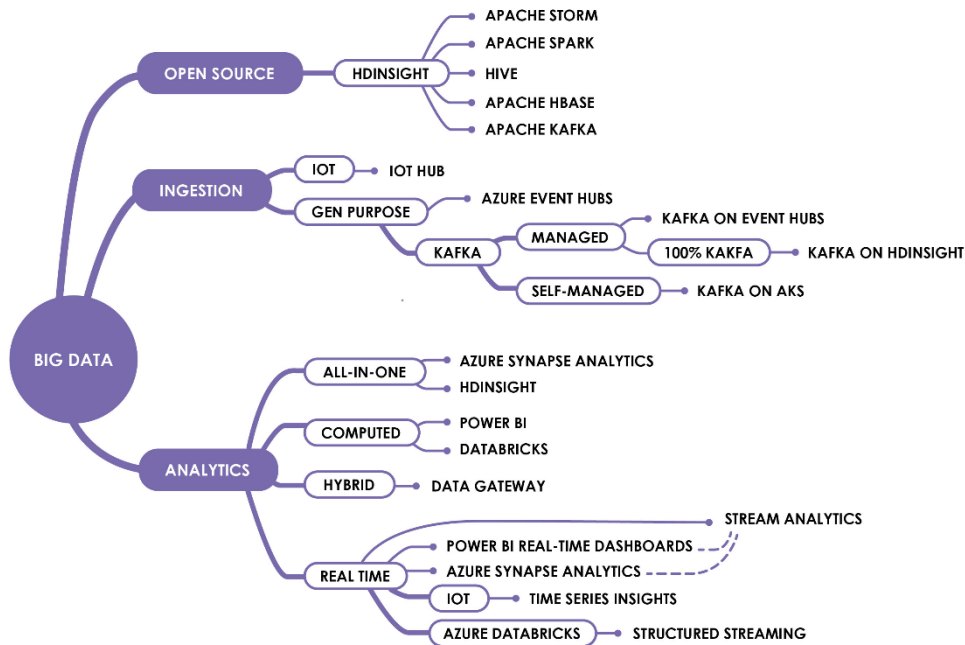


Figure 8.8 – Big data in Azure

We then covered ingesting big data, big data analytics, AI, machine learning, and deep learning. Our cross-cutting concerns included data migration, governance, and business-to-business data sharing. We ended with a near-real-time data streaming use case, which provided a finished simulator for you to browse. We included instructions for setting up your Power BI workspace, your Event Hubs instance, and your Stream Analytics instance.

Appeal

Make sure you try out our use case with Power BI. Once you see how easy it is to build and manage a dashboard, you'll start to imagine all the times you could leverage Power BI in your solutions.

Security architecture

Chapter 7, *Security Architecture*, introduced cloud-native security and took us on a tour of the security architecture map (see *Figure 8.9*):

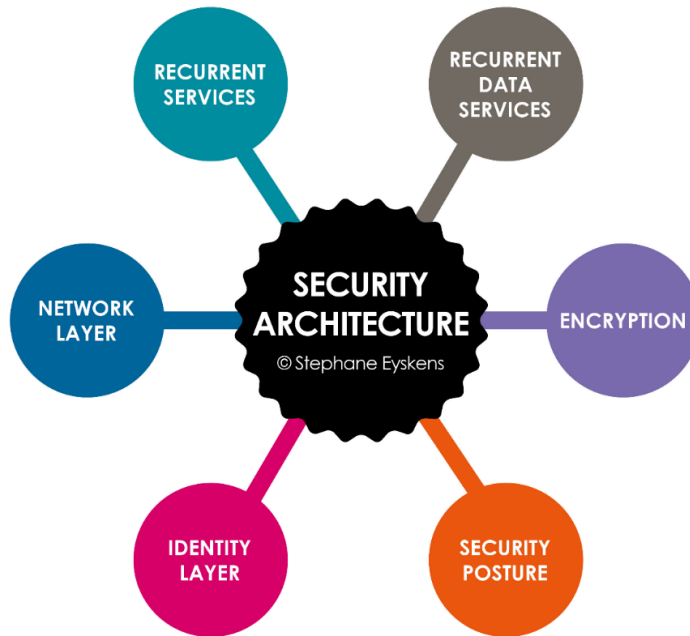


Figure 8.9 – The security architecture map

We discussed how to add security components to our Contoso use case. We drilled down into recurrent services, the network layer, the identity layer, recurrent data services (focusing on the security features), encryption (bring your own key, host your own key, and service-managed keys), and your security posture.

Appeal

Stacking network layers is not the best security approach in the cloud; it might give you a false impression of security. Remember that identity is by far the most important security layer!

Finally, *Chapter 8, Summary and Industry Scenarios*, made you smile a little (hopefully). Let's face it; after reading this crazy book, you kind of earned the chance to smile. You're still reading the chapter, so we won't spoil this chapter by summarizing it for you. Moving on...

Visiting the verticals

Next, we're going to browse some architectures for our industry verticals so that you can get a stronger understanding of how these architectures apply to your company, customers, and/or clients. The industries are listed in alphabetical order (we are a little strategically biased in what we're calling these verticals).

We warn you: you should expect to see a lot of links to the various industry-based architectures. Normally, this is a bad idea (to be sent outside the book) because you shouldn't have to leave a book in order to read it. However, we believe this is the best route to apply your learning, by testing it with many real customer architectures that go so deep that, well, if we wrote them ourselves, it would take another book. (But if that's what you want, it's duly noted!)

Automotive and transportation scenarios

Our three automotive and transportation scenarios build off the lessons we covered in *Chapter 6, Data Architecture*. All three of our architectures are fairly lightweight (what Microsoft currently refers to as **solution ideas**). They aren't deployable, but they can get your mind spinning down certain architectural paths to better understand your options. (We'll see some deployable architectures in our other verticals.) For our transportation scenarios, we have divided them into one that focuses on AI (predictive insights) and two that focus on analytics (predictive monitoring and IoT analytics).

Predictive insights with vehicle telematics

Our first architecture digs into applying AI to this industry, while our other two architectures take a differently flavored approach toward IoT analytics. For our AI scenario, we're looking to discover predictive insights on the health and driving habits of our vehicle. This impacts car dealerships, manufacturers, and even how insurance companies can leverage Azure to better understand their business. The diagnostic events are ingested through Azure Event Hubs, to Azure Stream Analytics, for Azure Machine Learning to create the core predictive analytics solutions. The data then moves through Blob storage (on to HDInsight and Data Factory), while being consumed through a constantly updated Power BI dashboard.

More information

You can further explore this Microsoft AI architecture for the automotive industry here:

Predictive Insights with Vehicle Telematics: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/predictive-insights-with-vehicle-telematics>

Predictive aircraft engine monitoring

Now for our two analytics scenarios. They are similar, but the focus is a little less on the machine learning aspects and a little more on straightforward analysis. Our predictive aircraft engine monitoring will analyze our aircraft telemetry to offer predictive maintenance. *Figure 8.10* demonstrates the architecture, which is currently available on Microsoft's Azure Architecture Center:

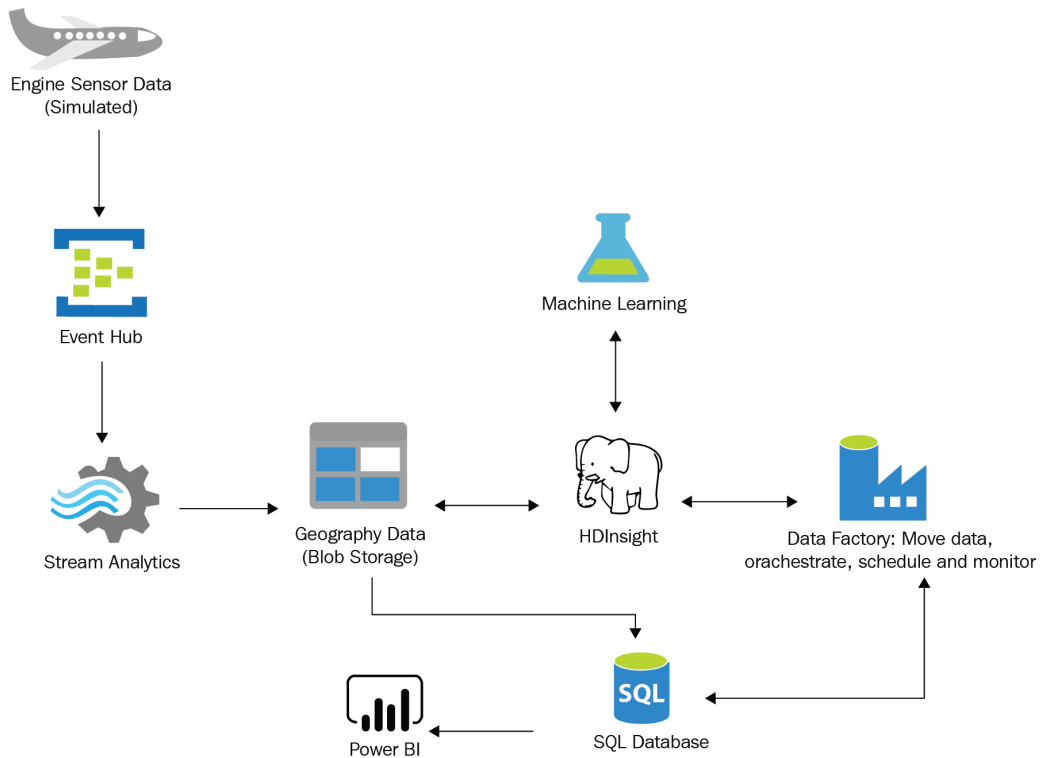


Figure 8.10 – The architecture for predictive aircraft engine monitoring

Azure Stream Analytics gives you as close to real-time analysis as you'll get. Azure Machine Learning predicts your system failures, and Azure SQL Database stores your prediction results, publishing them to your Power BI dashboard.

More information

You can further explore this lightweight Microsoft analytics architecture for the airline industry here:

Predictive Aircraft Engine Monitoring: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/aircraft-engine-monitoring-for-predictive-maintenance-in-aerospace>

IoT analytics for autonomous driving

For the IoT analytics architecture, Microsoft customers use it to analyze fast-flowing, high-volume streaming data. For example, Bosch uses this solution to determine real-time road conditions and current weather information in order to inform autonomous driving systems!

More information

You can further explore this lightweight Microsoft analytics architecture for the automotive industry here:

IoT analytics with Azure Data Explorer: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/iot-azure-data-explorer>

Banking and financial services scenarios

That brings us to the banking and financial services industries! Our first two architectures explore an infrastructure deployment scenario (banking system cloud transformation) and a blockchain scenario that provides on-demand compute (decentralized trust between banks).

Banking system cloud transformation

For our first architecture in this industry, *Banking system cloud transformation on Azure*, we're looking at a deployment that focuses on aspects of the solution's complete deployment ecosystem, which we first discussed in *Chapter 2, Solution Architecture*. The bank uses Docker to deliver the microservices containers to the Kubernetes cluster, which should be familiar as we saw a similar example in *Chapter 3, Infrastructure Design*.

This solution also leverages DevOps lessons learned in *Chapter 4, Infrastructure Deployment*, including important life cycle considerations, such as load testing and monitoring. It leverages the Azure container stack, third-party components (Docker, Grafana, and Prometheus), and open source tooling (Jenkins, KEDA, Apache JMeter, and Redis).

More information

Be sure to see the samples in the *Related resources* section at the bottom of the article:

Banking system cloud transformation on Azure: <https://docs.microsoft.com/azure/architecture/example-scenario/banking/banking-system-cloud-transformation>

Decentralized trust using blockchain

Our second architecture really brings to mind the lessons learned in *Chapter 3, Infrastructure Design*. This is a blockchain solution, *Decentralized trust between banks*. You leverage **Virtual Machine Scale Sets (VMSSes)** to provide on-demand compute. You store your private keys in Key Vault and use Load Balancer to spread out the requests.

More information

Be sure to try deploying the Ethereum **Proof-of-Authority (PoA)** blockchain demo that's mentioned at the bottom of the architecture article:

Decentralized trust between banks: <https://docs.microsoft.com/azure/architecture/example-scenario/apps/decentralized-trust>

Additional financial services architectures

In the following architectures, you will face AI scenarios (audit risk and business process management), security (fraud detection), and database (loan chargeoff and credit risk) scenarios.

More information

You can further explore these additional Microsoft architectures for the banking and financial service industries:

Auditing, risk, and compliance management: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/auditing-and-risk-compliance>

Business Process Management: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/business-process-management>

Real-time fraud detection: <https://docs.microsoft.com/azure/architecture/example-scenario/data/fraud-detection>

Loan ChargeOff Prediction with Azure HDInsight Spark Clusters: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/loan-chargeoff-prediction-with-azure-hdinsight-spark-clusters>

Loan ChargeOff Prediction with SQL Server: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/loan-chargeoff-prediction-with-sql-server>

Loan Credit Risk + Default Modeling: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/loan-credit-risk-analyzer-and-default-modeling>

Loan Credit Risk with SQL Server: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/loan-credit-risk-with-sql-server>

Gaming scenarios

For our gaming architectures, we're launching from our basic data discussions in *Chapter 6, Data Architecture*. We have three architectures that span two important gaming scenarios. First, we achieve low latency with a LAMP architecture. Then we'll look at two architectures that provide different perspectives (open source and full service) on data architecture for the gaming industry.

Low-latency multiplayer gaming

First, you'll explore the open source LAMP architecture, which stands for a specific stack: a **L**inux **U**buntu **V**M (**L**), **A**pache **w**eb **s**erver (**A**), **M**ySQL (**M**), and **P**HP (**P**). Azure Load Balancer gives the client the IP address from the DNS. The VMs read info from Azure Cache for Redis and read/write to/from Azure Database for MySQL.

More information

This architecture provides robust instructions on calculating the associated costs:

LAMP Gaming Reference Architectures: <https://docs.microsoft.com/gaming/azure/reference-architectures/general-purpose-lamp>

Gaming using MySQL or Cosmos DB

Then we have two sets of similar gaming architectures. The first is open source, leveraging MySQL and Azure HDInsight to address unpredictable traffic and offer low-latency gaming experiences with multiplayer capabilities. The second leverages Azure Cosmos DB, with Azure Databricks, Azure Functions, and Azure Notification Hub (providing the push notifications).

More information

Both architectures yield similar benefits of elastic scale:

Gaming using Azure Database for MySQL: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/gaming-using-azure-database-for-mysql>

Gaming using Cosmos DB: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/gaming-using-cosmos-db>

Healthcare scenarios

At the time of writing this book, we've been in lockdowns around the world, as we seek to fight (and survive) the deadly coronavirus. This means that today, healthcare is at the forefront of technology. Thus, Microsoft has plenty of new architectures for you to peruse!

Building a telehealth system on Azure

Our first two focus areas are in containers and storage, which build on our core lessons in *Chapter 3, Infrastructure Design*. Our container-focused architecture is *Building a telehealth system on Azure*. This solution provides remote hearing care services! The patient's information is stored in Azure Database for PostgreSQL, and AKS hosts the app's logic and allows optimal deployment. Azure Notification Hub is used to notify the patient of status and contact info, and Azure Functions is used to schedule all tasks.

More information

You'll find a lot of details for this architecture here:

Building a telehealth system on Azure: <https://docs.microsoft.com/azure/architecture/example-scenario/apps/telehealth-system>

Medical data storage architectures

For storage-focused architectures, *Medical Data Storage Solutions* ingests medical image data (via Azure Data Factory). The data is stored on Azure Blob storage and analyzed by the Azure Cognitive Services API. The AI results are stored in Azure Data Lake and consumed with a Power BI dashboard. Similarly, HIPAA and HITRUST compliant health data AI ingest the patient data into Azure Blob storage. Event Grid publishes the data to Azure Functions tasks to process it.

More information

Here are these two storage-oriented medical architectures:

Medical Data Storage Solutions: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/medical-data-storage>

HIPAA and HITRUST compliant health data AI: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/security-compliance-blueprint-hipaa-hitrust-health-data-ai>

AI healthcare solutions

In *Chapter 6, Data Architecture*, we touched on both AI and IoT scenarios.

More information

Take the time to browse through some of these AI healthcare solutions:

Implementing the Azure blueprint for AI:

<https://docs.microsoft.com/previous-versions/azure/industry-marketing/health/sg-healthcare-ai-blueprint>

Population Health Management for Healthcare: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/population-health-management-for-healthcare>

Predict Length of Stay and Patient Flow: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/predict-length-of-stay-and-patient-flow-with-healthcare-analytics>

Remote Patient Monitoring Solutions: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/remote-patient-monitoring>

Predicting length of stay using SQL Server R Services

This next architecture is very similar (in goal and final data consumption), but it's classified as an analytics scenario, as it uses SQL Server R Services.

More information

Review this solution idea:

Predicting Length of Stay in Hospitals: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/predicting-length-of-stay-in-hospitals>

Producing and consuming IoT healthcare data

When we covered IoT, we mostly showed you the perspective of ingesting the data provided by IoT systems. These architectures continue that theme by showing you various ways to produce and consume IoT data.

More information

Note that these architectures make a unique collection that represents how IoT solutions can have an impact during a pandemic:

Contactless IoT interfaces with Azure intelligent edge: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/contactless-interfaces>

COVID-19 Safe Solutions with IoT Edge: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/cctv-mask-detection>

IoT Connected Platform for COVID-19 detection and prevention: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/iot-connected-platform>

UVEN smart and secure disinfection and lighting: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/uv-en-disinfection>

Confidential computing on a healthcare platform

Finally, let's look at an architecture that ties into our topics from *Chapter 7, Security Architecture*. *Confidential computing on a healthcare platform* enables a health-based organization to secure financial data and patient information. The patient's information is stored in Azure Blob storage. The solution deploys code in an AKS confidential node (stored in a Redis cache). Azure Container Registry creates and manages the container image registry.

More information

See the *Deploy this scenario* section for information about how to practice implementing this solution:

Confidential computing on a healthcare platform: <https://docs.microsoft.com/azure/architecture/example-scenario/confidential/healthcare-inference>

Manufacturing scenarios

Can the cloud help you improve, build, and deliver your product? Absolutely. To understand how, let's explore a few key scenarios. We'll explore infrastructure (using IoT Hub and Azure Blockchain Workbench), industrial IoT analytics, and some important AI manufacturing scenarios.

Supply chain track and trace

For the manufacturing industry, let's start with a blockchain solution that builds on the lessons learned in *Chapter 3, Infrastructure Design*. *Supply Chain Track and Chase* provides IoT-enabled monitoring for a supply chain. If you were going to transport refrigerated goods, then you'd likely have compliance rules to follow (and contractual conditions that must be met), such as a temperature and humidity range.

More information

This solution begins in IoT Hub, and heavily leverages Azure Blockchain Workbench:

Supply Chain Track and Trace: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/supply-chain-track-and-trace>

Industrial IoT analytics

That moves us on to *Chapter 6, Data Architecture*, for the remaining architectures (moving from IoT to AI to analytics). *Azure Industrial IoT Analytics Guidance* provides a detailed tour of leveraging IoT systems for asset monitoring, processing dashboards, determining your **Overall Equipment Effectiveness (OEE)**, predictive maintenance, and forecasting.

More information

There are multiple articles in this content set:

Azure Industrial IoT Analytics Guidance: <https://docs.microsoft.com/azure/architecture/guide/iiot-guidance/iiot-architecture>

AI and analytics manufacturing architectures

For the remaining AI and analytics architectures (in our manufacturing vertical), the data is mostly ingested via Event Hubs, leveraging Azure Stream Analytics to provide near-real-time analysis on your input stream. Typically, Azure Machine Learning is then used to build the predictions.

More information

Please browse the following architectures:

Defect prevention with predictive maintenance: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/defect-prevention-with-predictive-maintenance>

Predictive Maintenance: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/predictive-maintenance>

Quality Assurance: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/quality-assurance>

Demand Forecasting for Shipping and Distribution: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/demand-forecasting-for-shipping-and-distribution>

Predictive Aircraft Engine Monitoring: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/aircraft-engine-monitoring-for-predictive-maintenance-in-aerospace>

Oil and gas scenarios

For our oil and gas architectures, we've got a nice variety that spans HPC, analytics, and IoT-focused scenarios. First, let's unravel the challenge of computing 3D modeling and seismic data.

Run reservoir simulation software on Azure

Our first scenario focuses heavily on HPC, which ties into what we learned in *Chapter 3, Infrastructure Design*. This architecture, *Run reservoir simulation software on Azure*, allows you to compute 3D reservoir modeling and visualize seismic data! This includes a sample INTERSECT simulation from Microsoft customer Schlumberger. The HB-series of high-performance VMs are deployed as a VMSS, which are multiple, identical VMs that are able to scale and complete high-performance compute tasks. Azure CycleCloud operates, manages, schedules, and deploys the big compute clusters.

More information

You can deploy this architecture using the example implementation:

Run reservoir simulation software on Azure: <https://docs.microsoft.com/azure/architecture/example-scenario/infrastructure/reservoir-simulation>

Oil and gas tank level forecasting

Next, we have a lighter, analytics-focused architecture. (Microsoft currently calls these basic architectures solution ideas.) Building off what you learned in *Chapter 6, Data Architecture, Oil and Gas Tank Level Forecasting* shows you how you'd prevent tank spillage and emergency shutdowns (as well as discover hardware malfunctions, schedule maintenance, detect leaks, and more). The data is ingested through Azure Event Hubs and analyzed in Azure Stream Analytics, and the forecasts are made in Azure Machine Learning. Azure Synapse Analytics stores the prediction results, which are then presented as visualizations in a Power BI dashboard!

More information

Continue reading about this lightweight architecture here:

Oil and Gas Tank Level Forecasting: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/oil-and-gas-tank-level-forecasting>

IoT monitor and manage loops

Finally, we continue what you learned in Data Architecture with a Microsoft IoT pattern, *IoT monitor and manage loops*. This pattern is very applicable for gas pipeline monitoring and to monitor and control the crude oil cracking process in an oil refinery.

More information

You can further explore this Microsoft architecture here:

IoT monitor and manage loops: <https://docs.microsoft.com/azure/architecture/example-scenario/iot/monitor-manage-loop>

Retail scenarios

Finally, we'll explore a robust library of architectures in the sister industry to manufacturing: retail. We'll start by browsing architectures that map back to *Chapter 6, Data Architecture*.

Retail and e-commerce Azure database architectures

First, let's browse some architectures that focus on database usage. All three scenarios are run through a browser that's built on Azure App Service (Web Apps). The logs and static catalog content are kept in Azure Storage. Then we have the main difference, whether the data catalog is organized in Azure Database for MySQL, Azure Database for PostgreSQL, or Azure Cosmos DB.

More information

You can browse the three options here:

Retail and e-commerce using Azure MySQL: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/retail-and-ecommerce-using-azure-database-for-mysql>

Retail and e-commerce using Azure PostgreSQL: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/retail-and-ecommerce-using-azure-database-for-postgresql>

Retail and e-commerce using Cosmos DB: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/retail-and-e-commerce-using-cosmos-db>

To determine which route to take, we think you'll want to make some higher-level design decisions.

More information

To further compare these database services (and many other options), see these Microsoft architectural guides (part of the Azure Data Architecture Guide):

Choosing a data storage technology in Azure: <https://docs.microsoft.com/azure/architecture/data-guide/technology-choices/data-storage>

Understand data store models: <https://docs.microsoft.com/azure/architecture/guide/technology-choices/data-store-overview>

Demand forecasting with Spark on HDInsight

The next step of progression for us involves three analytics-focused scenarios. These three demand forecasting architectures are similar in name and function, but they are distinct. The two *Price Optimization* versions show your data being initiated from your web app and/or web job. The data is then stored in Azure Blob storage or Azure Data Lake (depending on which scenario you follow). Spark on HDInsight ingests the raw data, to build the forecasting models. Azure Data Factory schedules your data flow, and a Power BI dashboard enables you to visualize your insights.

More information

You can explore these architectures here:

Demand Forecasting + Price Optimization: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/demand-forecasting-price-optimization-marketing>

Demand Forecasting and Price Optimization: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/demand-forecasting-and-price-optimization>

Demand forecasting with machine learning

This similar scenario (demand forecasting) follows a common pattern that we've seen previously, where Azure Event Hubs ingests the data, Stream Analytics analyzes it, Azure Machine Learning forecasts the demand, Azure SQL Database stores the prediction results, Azure Data Factory orchestrates and schedules, and then Power BI visualizes your data.

More information

You can find this machine learning architecture on the Azure Architecture Center:

Demand Forecasting: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/demand-forecasting>

AI retail scenarios

Our next step of progression is to move on to the AI scenarios. The first architecture (*Build a Real-time Recommendation API on Azure*) is fully deployable, while the other retail scenarios are lighter-weight solution ideas.

More information

Browse the following AI retail architectures:

Build a Real-time Recommendation API on Azure: <https://docs.microsoft.com/azure/architecture/reference-architectures/ai/real-time-recommendation>

Commerce Chatbot: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/commerce-chatbot>

Customer Feedback and Analytics: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/customer-feedback-and-analytics>

Personalized Offers: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/personalized-offers>

Personalized marketing solutions: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/personalized-marketing>

Predictive Marketing with Machine Learning: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/predictive-marketing-campaigns-with-machine-learning-and-spark>

Product recommendations for retail using Azure: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/product-recommendations>

Retail Assistant with Visual Capabilities: <https://docs.microsoft.com/azure/architecture/solution-ideas/articles/retail-assistant-or-vacation-planner-with-visual-capabilities>

Architecture for buy online, pick up in store

Lastly, we move on to a retail IoT scenario, *Buy online, pickup in store*. Due to the COVID-19 pandemic, a lot more customers have been picking up their items (including groceries) rather than shopping for the items themselves. This solution notifies the customer that the store has started packing the items and estimates when they will be available for pickup. The solution also uses Azure Maps to identify the customer's current location, as well as video analytics to obtain the customer's license plate details (which lets the store know when the customer pulls up into the parking lot).

More information

You'll find the solution details on the Azure Architecture Center:

Retail - Buy online, pickup in store (BOPIS): <https://docs.microsoft.com/azure/architecture/example-scenario/iot/vertical-buy-online-pickup-in-store>

The unique values of this book

The truth is that there are many other books about architecting cloud solutions. You'll also find a lot of online articles on the subject (as further shown by the architecture solutions we have listed in this chapter). However, in this book, Stephane has brought to life his unique map-based approach to exploring and learning Azure architectures. Based on our success in community and the community's response to his architectures, we believe this unique approach helps bring the architectures to life.

When we first put this book together, we thought of a few unique qualities that we think make this book stand out. Although you've read the book, understanding these qualities might help refresh your memory, as you refer back to certain chapters and sections in the book. Likewise, you could spend a good amount of time browsing and reviewing the online resources and the map/diagram files that we have for you. Thus, as you continue to leverage this book, we hope that its key differentiators resonate with you:

- **Map-based:** We could have called this an *atlas*. *Mapbook* actually isn't a word. We made it up. So, I suppose it is a word now (as all words are made up). But you don't have to guess what a mapbook is... it's a book full of maps. And, you'll find even more maps on our GitHub repo (or the same/similar maps with more details that are more easily consumed outside of a book). Take a look at the architectures we feature in this chapter. That's usually the closest you can get to map-based learning. We owe this perspective and vision to the original community efforts of Microsoft MVP Stéphane Eyskens. When you know something works, you try going big with it, and having seen the community's enthusiasm for map-based architectures, we feel like we went big with this map-based architecture book.
- **Expansive:** Not only did we go big with maps, but we also went big with a high-level approach, taken from a variety of perspectives: solutions, infrastructure, deployment, development, data, and security. Hopefully, this approach leads to new ways of thinking and opportunities for everyone who reads it.

- **Azure-specific:** Azure-specific cloud architecture resources are not as common as you'd think. Most architectural resources (for cloud platforms) tend to be generic (or more development - and implementation-oriented), and thus they aren't quite as straightforward or applicable to building your architecture.

But here's the beauty of these differences: because *The Azure Cloud-Native Architecture Mapbook* is so unique, most any other architecture and development book (and online resource) can be used as a complement to this book. You can read this book to get a high-level perspective, and then you can read another book (or online architectural guidance) to dig into a specific scenario.

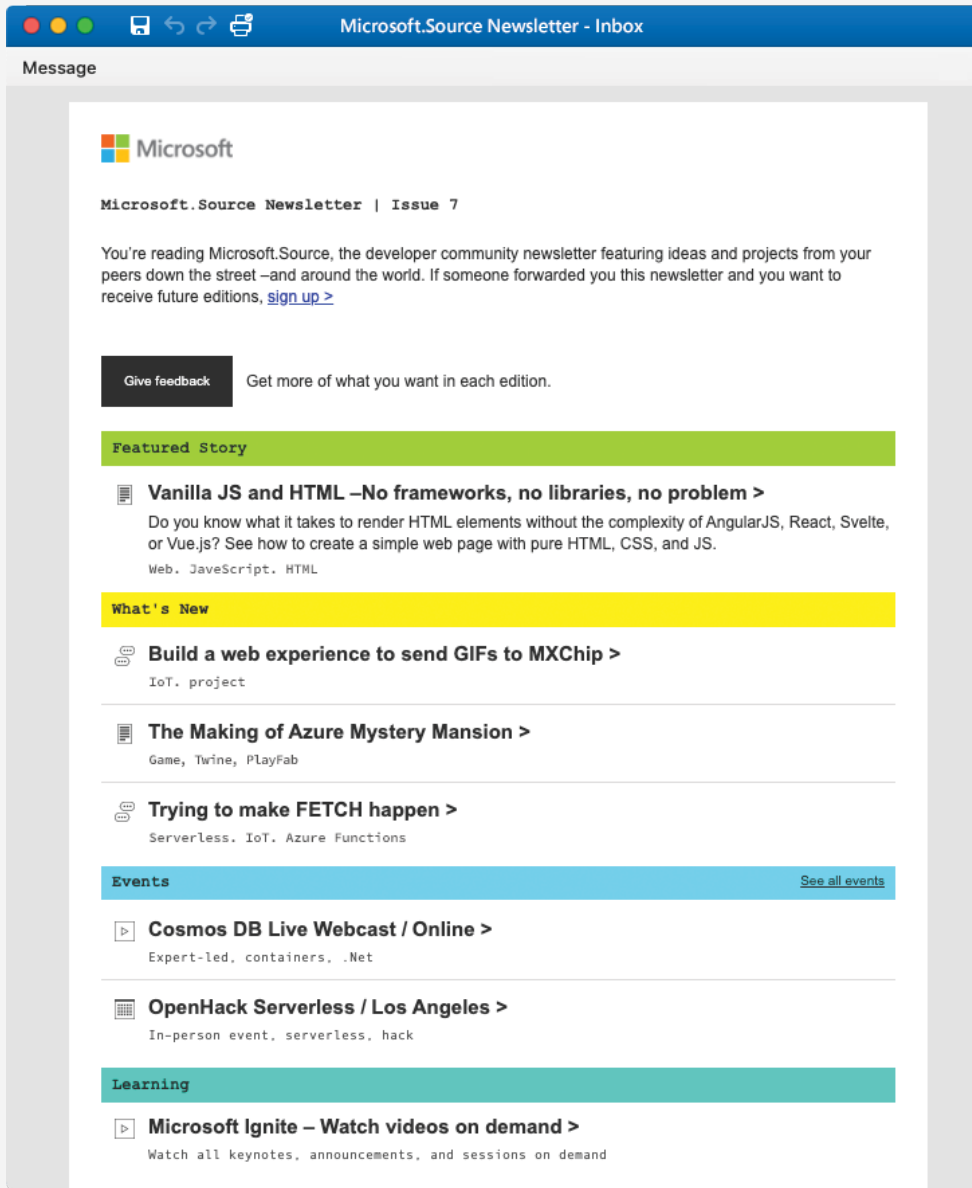
So, what did you gain from reading this book?

Summary

Through the course of this book, you gained a high-level view of what common patterns of architectures might look like. Hopefully, while reading *The Azure Cloud-Native Architecture Mapbook*, you thought of scenarios such as those in this chapter, an actual architecture that meets your business needs (or at least an applicable sample). Even if you're a seasoned architect, we suspect that you gained new perspectives on how to architect your solutions (and perhaps you picked up a few tips and tricks). Ultimately, we hope that you observed how you can strengthen your organizational weaknesses, bringing new values to your companies, customers, and/or clients.

In summary, you gained an overall architectural knowledge (or review) of the Microsoft Azure cloud platform. You dug deeper into the possibilities of building a full Azure solution, learned best practices for designing and deploying an Azure infrastructure, and reviewed patterns and possibilities for building a complete solution.

The more you put into your learning from this book, the architectures we recommend, and exploring additional books and resources, the more you are bound to grow in your abilities to impact your projects and ultimately the businesses you contribute to. We'll be the first to congratulate you, but you should be next. You should reflect on your journey and be proud of what you can accomplish.



By developers, for developers

Microsoft.Source newsletter

Get technical articles, sample code, and information on upcoming events in Microsoft.Source, the curated monthly developer community newsletter.

- Keep up on the latest technologies
- Connect with your peers at community events
- Learn with hands-on resources



Sign up



Packt . com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [packt . com](http://packt.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [customercare@packtpub . com](mailto:customercare@packtpub.com) for more details.

At [www . packt . com](http://www.packt.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

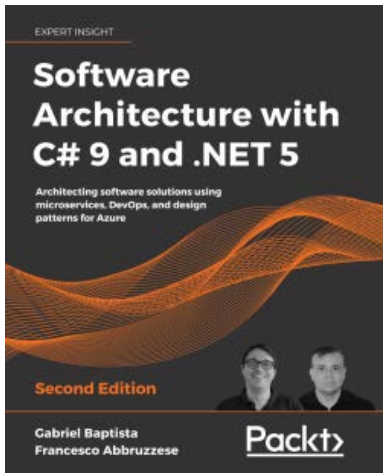


Modern Computer Architecture and Organization

Jim Ledin

ISBN: 978-1-83898-439-7

- Get to grips with transistor technology and digital circuit principles
- Discover the functional elements of computer processors
- Understand pipelining and superscalar execution
- Work with floating-point data formats
- Understand the purpose and operation of the supervisor mode
- Implement a complete RISC-V processor in a low-cost FPGA
- Explore the techniques used in virtual machine implementation
- Write a quantum computing program and run it on a quantum computer



Software Architecture with C# 9 and .NET 5 - Second Edition

Gabriel Baptista, Francesco Abbruzzese

ISBN: 978-1-80056-604-0

- Use different techniques to overcome real-world architectural challenges and solve design consideration issues
- Apply architectural approaches such as layered architecture, service-oriented architecture (SOA), and microservices
- Leverage tools such as containers, Docker, Kubernetes, and Blazor to manage microservices effectively
- Get up to speed with Azure tools and features for delivering global solutions
- Program and maintain Azure Functions using C# 9 and its latest features
- Understand when it is best to use test-driven development (TDD) as an approach for software development
- Write automated functional test cases
- Get the best of DevOps principles to enable CI/CD environments

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

access: 13
accounts: 16
acquainted: 3, 12
acquire: 11
acquiring: 21
acronym: 16
active: 20
activities: 14, 23
additional: 12, 17
add-ons: 15
address: 11, 18, 24
adobe: 16
adopters: 23
advance: 15
advantages: 12
algorithm: 17
allocation: 15-16
analysis: 20
analytics: 6
analyze: 20
api-driven: 14
applicable: 18, 22
archimate: 4
architect: 3-4, 7-9, 11-12, 16, 18-19, 22, 25
artifacts: 22

artificial: 6, 21
aspects: 10, 19, 22-23
assets: 4, 10, 13, 15, 20-21
assistants: 21
associated: 3, 14, 17
attention: 11
audience: 19
audit: 21-22, 24
auditors: 22
automation: 24
azure: 3-7, 10-12, 14-20, 22, 25

B

backend: 9
background: 11
backups: 16
benefits: 6, 16
breaking: 11
bridge: 10, 13
browser: 9
budgets: 20
build: 5, 11, 13, 24-25
building: 4, 10, 15, 21-22, 24
built-in: 11, 13, 20, 24
business: 4, 6, 10-11, 13, 16, 19-24

C

 caching: 11
 capacity: 12, 15
 cases: 3, 12, 14, 17
 catalog: 7, 17-18
 classify: 17
 cloud: 3, 5, 8-16, 18-25
 cluster: 7
 cobit: 22
 coding: 10
 company: 4, 14-16, 19-21, 23
 compliance: 8
 components: 6-7, 17
 connection: 18
 connects: 18, 22
 consumer: 12, 14-15, 21, 24
 containers: 15-16
 contoso: 21-22, 24
 control: 15, 22, 24
 costs: 12-15, 22, 25
 customer: 17, 21

D

 database: 9, 16
 databricks: 16
 dbaas: 16
 deep-dive: 4
 default: 20
 defining: 4, 10, 19-20, 23
 deploy: 24
 deployment: 16
 design: 10, 15-16, 21, 24
 designing: 5-6, 20, 23
 develop: 21, 24-25
 developers: 7, 11
 developing: 23
 devops: 7

 digital: 13, 19-21
 directory: 20
 domain: 5, 11, 18
 domains: 18
 drivers: 19-20, 22-25
 dynamics: 16

E

 elastic: 14, 16
 elasticity: 15
 enablers: 22-23
 encrypted: 9
 encryption: 9
 enterprise: 4-5, 11, 19, 25
 equivalent: 8, 13
 exception: 17, 24
 executive: 20, 22
 exercise: 17, 20, 22
 explore: 6, 9-11, 16

F

 factors: 3, 19, 21, 25
 features: 10-12, 15, 22
 fictitious: 25
 field: 5, 8
 focus: 3, 6, 9-10, 15, 22
 focused: 11, 24
 framework: 4, 21-22
 frameworks: 5, 22-23
 front: 7, 16, 19-20, 22

G

 gateway: 9, 11
 github: 4
 given: 4-5, 18, 24

H

hardware: 13, 19
high-level: 5-7
highlight: 17
https: 4-5, 10, 17, 22
hybrid: 10, 13

I

idaas: 16
integrate: 10, 20
internal: 15, 21
isaca: 22

K

kubernetes: 7, 15

L

landscape: 4, 6, 11, 21, 25
launch: 19, 21-22
launching: 13, 24
layers: 14, 20
learning: 17, 20
local: 13
lock-in: 15
logic: 14, 18

M

machine: 15, 17
mainstream: 15
managed: 9, 13, 15-16, 21
management: 20-21, 23
market: 13, 19, 21-22, 24
master: 4-5, 18

microsoft: 3, 5, 10, 16-18, 22
migrating: 15
modeling: 4, 6
models: 3, 11-12, 16-17, 19, 24-25
modern: 11, 16, 21

N

network: 9, 15
nodes: 15
non-cloud: 9

O

offload: 11
operates: 23
operations: 12-16, 25
order: 6, 10, 13-14, 16, 20
os-patched: 15
outsourced: 13-14
owasp: 9
owner: 23
ownership: 13

P

perform: 14
performed: 20
perimeter: 9, 15
platform: 3-4, 7, 11, 13, 16, 20-24
practices: 10, 13, 16, 21, 23-24
pre-paid: 12, 16
pre-pays: 14
primary: 10, 17, 24
principles: 20-21, 23-25
private: 5
processes: 10, 13-14, 21, 23
processing: 16

products: 17, 20-22
profiles: 4, 25
program: 23
protect: 9
provider: 12-14, 16, 19, 24
public: 5, 8

R

real-world: 17, 23, 25
rebooting: 15
recovery: 13, 16
refactor: 13, 15
resources: 12, 15, 22
reviewed: 8, 17, 21, 25
roadmaps: 5
ruleset: 9
runtime: 8
rushing: 19

S

salesforce: 16
sandbox: 20
scale: 14, 19
scaling: 11
security: 5, 8-9, 15, 20, 24
semantics: 18
serverless: 14-16
servers: 13
services: 6-7, 10-11, 13, 16-19, 21-24
setting: 10, 20
setup: 10
solutions: 4-5, 11, 13, 17-18, 20-21, 24
source: 9-10, 22
storage: 6, 14, 16
stores: 8, 16

strategy: 13, 19-25
structured: 20
systems: 8, 10-11, 13, 21

T

technical: 4, 7-8, 11, 20
techniques: 16
technology: 4, 7, 22-23
togaf: 4-5
toolchain: 24
tools: 20-21
transit: 9
transition: 15, 20
trigger: 14
t-shaped: 5

U

upgrades: 15
usage: 15

V

validation: 11
value: 6, 14, 25
vendor: 15, 24
version: 15
versus: 11, 13, 24
viewpoint: 4
virtual: 15, 23

W

warehouses: 7
workloads: 10, 13-15