



*The Addison-Wesley Signature Series*

A VAUGHN VERNON SIGNATURE  
BOOK

# CONTINUOUS ARCHITECTURE IN PRACTICE

## SOFTWARE ARCHITECTURE IN THE AGE OF AGILITY AND DEVOPS

---

MURAT ERDER  
PIERRE PUREUR  
EOIN WOODS

*Foreword by* KURT BITTNER



# **Continuous Architecture in Practice**

**Software Architecture in the Age of Agility and DevOps**

**Murat Erder  
Pierre Pureur  
Eoin Woods**

 Addison-Wesley

# Foreword

Viewed from a sufficiently great distance, Earth looks serene and peaceful, a beautiful arc of sea and cloud and continents. The view at ground level is often anything but serene; conflicts and messy tradeoffs abound, and there are few clear answers and little agreement on the path forward.

Software architecture is a lot like this. At the conceptual level presented by many authors, it seems so simple: apply some proven patterns or perspectives, document specific aspects, and refactor frequently, and it all works out. The reality is much messier, especially once an organization has released something and the forces of entropy take over.

Perhaps the root problem is our choice of using the “architecture” metaphor; we have a grand idea of the master builder pulling beautiful designs from pure imagination. In reality, even in the great buildings, the work of the architect involves a constant struggle between the opposing forces of site, budget, taste, function, and physics.

This book deals with the practical, day-to-day struggles that development teams face, especially once they have something running. It recognizes that software architecture is not the merely conceptual domain of disconnected experts but is the rough-and-tumble give-and-take daily tussle of team members who have to balance tradeoffs and competing forces to deliver resilient, high-performing, secure applications.

While balancing these architectural forces is challenging, the set of principles that the authors of this book describe help to calm the chaos, and the examples that they use help to bring the principles to life. In doing so, their book bridges the significant gap between the Earth-from-orbit view and the pavement-level view of refactoring microservice code.

Happy architecting!

*—Kurt Bittner*

# Introduction

It has been a few years since we (Murat and Pierre) published *Continuous Architecture*,<sup>1</sup> and much has changed in that time, especially in the technology domain. Along with Eoin Woods, we therefore set out to update that book. What started as a simple revision, however, became a new book in its own right: *Continuous Architecture in Practice*.

<sup>1</sup> Murat Erder and Pierre Pureur, *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World* (Morgan Kaufmann, 2015).

While *Continuous Architecture* was more concerned with outlining and discussing concepts, ideas, and tools, *Continuous Architecture in Practice* provides more hands-on advice. It focuses on giving guidance on how to leverage the continuous architecture approach and includes in-depth and up-to-date information on topics such as security, performance, scalability, resilience, data, and innovation.

We revisit the role of architecture in the age of agile, DevSecOps, cloud, and cloud-centric platforms. We provide technologists with a practical guide on how to update classical software architecture practice in order to meet the complex challenges of today's applications. We also revisit some of the core topics of software architecture: the role of the architect in the development team, meeting stakeholders' quality attribute needs, and the importance of architecture in achieving key cross-cutting concerns, including security, scalability, performance, and resilience. For each of these areas, we provide an updated approach to making the architecture practice relevant, often building on conventional advice found in the previous generation of software architecture books and explaining how to meet the challenges of these areas in a modern software development context.

*Continuous Architecture in Practice* is organized as follows:

In [Chapter 1](#), we provide context, define terms, and provide an overview of the case study that will be used throughout each

chapter (more details for the case study are included in [Appendix A](#)).

In [Chapter 2](#), our key ideas are laid out, providing the reader with an understanding of how to perform architecture work in today's software development environment.

In [Chapters 3](#) through [7](#), we explore a number of architecture topics that are central to developing modern applications: data, security, scalability, performance, and resilience. We explain how software architecture, in particular the Continuous Architecture approach, can help to address each of those architectural concerns while maintaining an agile way of working that aims to continually deliver change to production.

In [Chapters 8](#) and [9](#), we look at what is ahead. We discuss the role of architecture in dealing with emerging technologies and conclude with the challenges of practicing architecture today in the era of agile and DevOps as well as potential ways to meet those challenges.

We expect some of our readers to be software architects who understand the classical fundamentals of the field (perhaps from a book such as *Software Architecture in Practice*<sup>2</sup> or *Software Systems Architecture*<sup>3</sup>) but who recognize the need to update their approach to meet the challenges of today's fast-moving software development environment. The book is also likely to be of interest to software engineers who want to learn about software architecture and design and who will be attracted by our practical, delivery-oriented focus.

<sup>2</sup> Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice* (Addison-Wesley, 2012).

<sup>3</sup> Nick Rozanski and Eoin Woods, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives* (Addison-Wesley, 2012).

To keep the scope of this book manageable and focused on what has changed since our last book, we assume that readers are familiar with the basics of mainstream technical topics such as information security, cloud computing, microservice-based architecture, and common automation

techniques such as automated testing and deployment pipelines. We expect that our readers are familiar with the fundamental techniques of architectural design, how to create a visual model of their software, and associated techniques such as the domain-driven design (DDD) approach.<sup>4</sup> For those who feel unsure about architectural design fundamentals, we suggest starting with a well-defined approach such as the Software Engineering Institute's attribute-driven design<sup>5</sup> or a simpler approach such as the one outlined in chapter 7 of *Software Systems Architecture*. Software modeling, although neglected for a few years, seems to be returning to mainstream practice. For those who missed it first time around, chapter 12 of *Software Systems Architecture* provides a starting point, and Simon Brown's books<sup>6,7</sup> are a more recent and very accessible introduction to it.

<sup>4</sup> For more information on DDD, please see Vaughn Vernon, *Implementing Domain-Driven Design* (Addison-Wesley, 2013).

<sup>5</sup> Humberto Cervantes and Rick Kazman, *Designing Software Architectures: A Practical Approach* (Addison-Wesley, 2016). The AAD approach is also outlined in Bass, Clements, and Kazman, *Software Architecture in Practice*, chapter 17s.

<sup>6</sup> Simon Brown, *Software Architecture for Developers: Volume 1—Technical Leadership and the Balance with Agility* (Lean Pub, 2016). <https://leanpub.com/b/software-architecture>

<sup>7</sup> Simon Brown, *Software Architecture for Developers: Volume 2—Visualise, Document and Explore Your Software Architecture* (Lean Pub, 2020). <https://leanpub.com/b/software-architecture>

The other foundational architecture practice that we don't discuss in this book is how to assess a software architecture. This topic is covered in chapter 6 of our previous book, chapter 14 of *Software Systems Architecture*, and chapter 21 of *Software Architecture in Practice*. You can also find a lot of information about architectural evaluation methods such as the Architecture Tradeoff Analysis Method (ATAM) via an Internet search.

We also assume an existing knowledge of agile development and so do not provide in-depth discussions of software development life cycle processes such as agile, Scrum, and the Scaled Agile Framework (SAFe), nor do we discuss software deployment and operation approaches, such as DevSecOps, in any depth. We deliberately do not include details on any specific technology domain (e.g., database, security, automation). We of course refer to these topics where relevant, but we assume our readers are

generally familiar with them. We covered these topics, except for technology details, in *Continuous Architecture*. Also, please note that terms defined in the glossary are highlighted in **bold** the first time they appear in this book.

The foundations of software architecture haven't changed in the last four years. The overall goal of architecture remains to enable early and continuous delivery of business value from the software being developed. Unfortunately, this goal isn't always prioritized or even understood by many architecture practitioners.

The three of us call ourselves architects because we believe there is still no better explanation of what we do every day at work. Throughout our careers covering software and hardware vendors, management consultancy firms, and large financial institutions, we have predominantly done work that can be labeled as software and enterprise architecture. Yet, when we say we are architects, we feel a need to qualify it, as if an explanation is required to separate ourselves from the stereotype of an IT architect who adds no value. Readers may be familiar with an expression that goes something like this: "I am an architect, but I also deliver/write code/engage with clients \_\_\_\_\_ [fill in your choice of an activity that is perceived as valuable]."

No matter what the perception, we are confident that architects who exhibit the notorious qualities of abstract mad scientists, technology tinkerers, or presentation junkies are a minority of practitioners. A majority of architects work effectively as part of software delivery teams, most of the time probably not even calling themselves architects. In reality, all software has an architecture (whether or not it is well understood), and most software products have a small set of senior developers who create a workable architecture whether or not they document it. So perhaps it is better to consider architecture to be a skill rather than a role.

We believe that the pendulum has permanently swung away from conventional software architectural practices and perhaps from conventional enterprise architecture as well. However, based on our collective experience, we believe that there is still a need for an architectural approach that can encompass agile, continuous delivery, DevSecOps, and cloud-centric computing, providing a broad architectural perspective to unite and integrate these approaches to deliver against our

business priorities. The main topic of this book is to explain such an approach, which we call Continuous Architecture, and show how to effectively utilize this approach in practice.

# **Contents**

**Foreword**

**Introduction**

**Chapter 1. Why Software Architecture Is More Important than Ever**

**Chapter 2. Architecture in Practice: Essential Activities**

**Chapter 3. Data Architecture**

**Chapter 4. Security as an Architectural Concern**

**Chapter 5. Scalability as an Architectural Concern**

**Chapter 6. Performance as an Architectural Concern**

**Chapter 7. Resilience as an Architectural Concern**

**Chapter 8. Software Architecture and Emerging Technologies**

**Chapter 9. Conclusion**

**Appendix A. Case Study**

**Appendix B. Comparison of Technical Implementations of Shared  
Ledgers**

**Glossary**

# Table of Contents

## Foreword

## Introduction

### Chapter 1. Why Software Architecture Is More Important than Ever

What Do We Mean by Architecture?

Software Industry Today

Current Challenges with Software Architecture

Software Architecture in an (Increasingly) Agile World

Introducing Continuous Architecture

Applying Continuous Architecture

Introduction to the Case Study

Summary

### Chapter 2. Architecture in Practice: Essential Activities

Essential Activities Overview

Architectural Decisions

Quality Attributes

Technical Debt

Feedback Loops: Evolving an Architecture

Common Themes in Today's Software Architecture Practice

Summary

### Chapter 3. Data Architecture

Data as an Architectural Concern

Key Technology Trends

Additional Architectural Considerations

[Summary](#)

[Further Reading](#)

## **Chapter 4. Security as an Architectural Concern**

[Security in an Architectural Context](#)

[Architecting for Security](#)

[Architectural Tactics for Mitigation](#)

[Maintaining Security](#)

[Summary](#)

[Further Reading](#)

## **Chapter 5. Scalability as an Architectural Concern**

[Scalability in the Architectural Context](#)

[Architecting for Scalability: Architecture Tactics](#)

[Summary](#)

[Further Reading](#)

## **Chapter 6. Performance as an Architectural Concern**

[Performance in the Architectural Context](#)

[Architecting for Performance](#)

[Summary](#)

[Further Reading](#)

## **Chapter 7. Resilience as an Architectural Concern**

[Resilience in an Architectural Context](#)

[Architecting for Resilience](#)

[Architectural Tactics for Resilience](#)

[Maintaining Resilience](#)

[Summary](#)

[Further Reading](#)

## **Chapter 8. Software Architecture and Emerging Technologies**

Using Architecture to Deal with Technical Risk Introduced by New Technologies

Brief Introduction to Artificial Intelligence, Machine Learning, and Deep Learning

Using Machine Learning for TFX

Using a Shared Ledger for TFX

Summary

Further Reading

## **Chapter 9. Conclusion**

What Changed and What Remained the Same?

Updating Architecture Practice

Data

Key Quality Attributes

The Architect in the Modern Era

Putting Continuous Architecture in Practice

## **Appendix A. Case Study**

Introducing TFX

The Architectural Description

Other Architectural Concerns

## **Appendix B. Comparison of Technical Implementations of Shared Ledgers**

## **Glossary**

# Chapter 1. Why Software Architecture Is More Important than Ever

*Continuous improvement is better than delayed perfection.*

—Mark Twain

While the goal of architecture remains to deliver business value, the increasing speed of delivery expected from information technology (IT) practitioners within enterprises presents new challenges. At the same time, the ease-of-use and 24/7 expectations of end users are being driven by the overwhelming expansion of technology in our daily lives—we have moved from PCs to tablets to smartphones to wearable technology. Computers have crawled out of the labs and are now nestled in our pockets. They are almost always connected to each other, and their capabilities are surpassing our wildest expectations. Today's software delivery teams are now expected to operate at Internet and cloud time and scale. This has significantly increased the demands from business stakeholders and resulted in the widening adoption of agile, continuous delivery, and DevOps practices.

To meet these challenges and fulfill the objective of delivering business value in this fast-paced environment, we believe that it is more important than ever for architecture activities to enable early and **continuous delivery** of value and ensure that what they do supports this goal.

## What Do We Mean by Architecture?

When we talk about architecture, we are concerned with software architecture. But how can we define *software architecture*? Let us look at a few common definitions.

According to the portal maintained by the International Federation for Information Processing (IFIP) Working Group 2.10 on Software Architecture:<sup>1</sup>

<sup>1</sup> International Federation for Information Processing (IFIP) Working Group 2.10 focuses on software architecture and maintains the Software Architecture Portal at <http://www.softwarearchitectureportal.org>

*Software architecture refers to the fundamental structures of a software system and the discipline of creating such structures and systems. Each structure comprises software elements, relations among them, and properties of both elements and relations. The architecture of a software system is a metaphor, analogous to the architecture of a building. It functions as a blueprint for the system and the developing project, laying out the tasks necessary to be executed by the design teams.*

*Software architecture is about making fundamental structural choices that are costly to change once implemented. Software architecture choices include specific structural options from possibilities in the design of the software. . . .*

*Documenting software architecture facilitates communication between stakeholders, captures early decisions about the high-level design, and allows reuse of design components between projects*

The International Standards Organization and Institute of Electrical and Electronics Engineers (IEEE) uses the following definition:<sup>2</sup>

<sup>2</sup> This definition is presented in ISO/IEC/IEEE 42010:2011, *Systems and Software Engineering—Architecture Description* (last reviewed and confirmed in 2017).  
<https://www.iso.org/standard/50508.html>

*Architecture: Fundamental concepts or properties of a system in its environment embodied in its elements, their relationships, and in the principles of its design and evolution.*

Specifically, in this book, we deal with the technical concerns of software development within a commercial enterprise.

The four most common reasons for developing a software architecture today are to

1. Achieve quality attribute requirements for a software system.  
Architecture is concerned with implementing important quality

attribute requirements, such as security, scalability, performance, and resiliency.

2. Define the guiding principles and standards for a project or product and develop blueprints. Architecture is a vision of the future and supporting tools to help communicate the vision. Blueprints enable the architecture to be abstracted at an appropriate level to make business and technical decisions. They also facilitate the analysis of properties and comparison of options.
3. Build usable (and perhaps even reusable) services. A key aspect of software architecture is defining good interfaces for services.
4. Create a roadmap to an IT future state. Architecture deals with transition planning activities that lead to the successful implementation of an IT blueprint.

Achieving these goals, while building a large software system in an effective and efficient manner, traditionally involves creating a blueprint—commonly called a **software architecture**. Such an architecture usually includes descriptive models (defined in both business and technology terms) to aid decision makers in understanding how the organization operates today and how it wants to operate in the future and to help the project or product team successfully transition to the desired technology state.

However, there are exceptions to this approach. Some large and successful companies believe that they do not need to follow this approach because large blueprints are hard to maintain and provide no value if allowed to become outdated. Instead, those companies focus on standards and common services, as they believe that those are more valuable in the long run. How to balance the traditional blueprint view with this approach is exactly the purpose of the book.

Now that we have a working definition of software architecture, let's look at it in the larger context of today's software industry.

## Software Industry Today

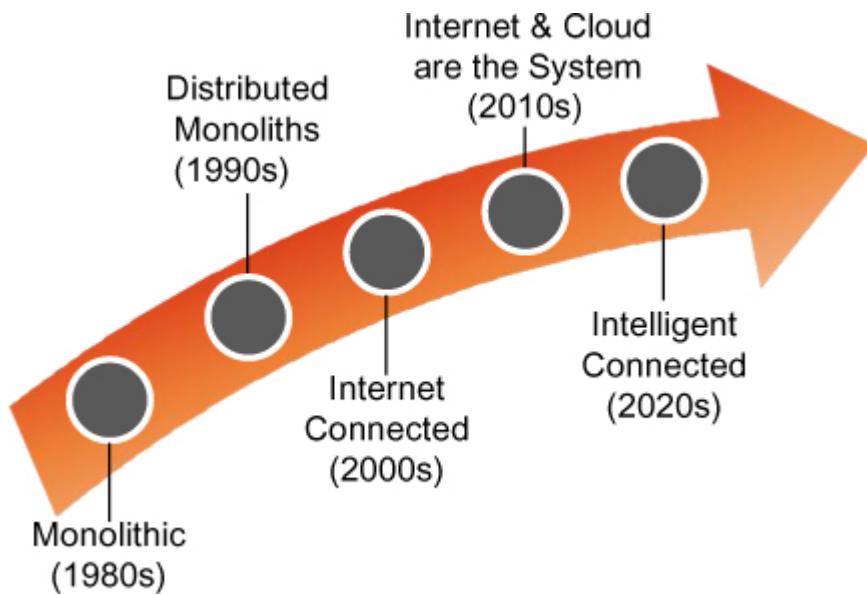
It has been said that today “every company is a software company,”<sup>3</sup> and though it overstates the point, this statement captures the reality that most organizations need to learn how to make their products and services

relevant in the Internet age. Doing so normally involves developing—and constantly changing—complex software systems.

<sup>3</sup> Satya Nadella, Microsoft CEO, as quoted in João Paulo Carvalho, *Every Business Is a Software Business*, Digital Transformation@Quidgest. <https://quidgest.com/en/articles/every-business-software-business>

We used to develop software in a relatively predictable way, seeing the process as a project with a fairly fixed set of requirements and a period during which the software was developed and delivered. In this era, we saw a lot of the key design decisions being made at the beginning of the process, and we didn't expect a lot to change during development.

However, over the past 40 years, technology changes have driven significant changes in the way we need to develop software applications (see [Figure 1.1](#)).



**Figure 1.1** Five ages of software systems

Prior to and during the 1980s, we were largely in the monolithic era of information systems. A software application tended to run on a single machine and to use an entire system software environment (screen manager, transaction monitor, database, operating system) supplied by the manufacturer of the machine. The software application itself tended to be seen as a single monolithic entity. Software architecture was largely

dictated by the computer manufacturer rather than by the end-user software developers.

We then moved into the client/server era. We started splitting our applications into two or three physical tiers driven by the widespread availability of desktop PCs and UNIX-based server computers. Software architecture wasn't very complicated, but we were starting to have to make decisions for ourselves: Where did the business logic live? In the user interface? In database-stored procedures? In both?

Later, in the 2000s, software architecture really came of age as the Internet expanded to offer new opportunities and capabilities, and we needed to connect our systems to it and have users from outside our organization, potentially anywhere in the world, use the system. Security, performance, scalability, and availability suddenly took on a new importance and complexity, and the industry realized the importance of quality attributes and architectural decisions.

More recently, the Internet has become part of our daily lives, and cloud computing, Internet-connected services, and the API economy all mean that the Internet is part of our systems rather than a service our systems are connected to. In turn, this expansion of the Internet has driven the need for rapid change and flawless quality attributes, leading to trends such as the use of public cloud computing, microservices, and continuous delivery.

We cannot be sure what the future holds, but it seems likely that current trends will continue and our systems will be ever-more connected and will need to exhibit the sort of pseudo-intelligence that Google and Facebook are building into their consumer-facing applications today, leading to yet more demands on our software architectures.

As we progressed through these periods and needed to respond to the demand for faster, more reliable software delivery, we quickly found the limitations of our classical, up-front planning approach to software delivery, which led to the development of agile methods of software delivery that embrace change throughout the software life cycle. The problem often then became one of deployment because it could take months to commission new infrastructure. The need for faster deployment led to the adoption of cloud computing, which removes this bottleneck.

Once we could create and change our infrastructure at will, the focus moved back to the software deployment process, which was often manual and error prone. Efforts to improve the process gave rise to the adoption of **container** technologies and a high degree of deployment automation. In principle, using container technology and deployment automation allowed a rapid flow of changes to production. Many operations groups, however, were unable to absorb change at this rate, as they needed to coordinate manual release checking by a large number of siloed specialists, thereby slowing the process again. Consequently, **DevOps** and DevSecOps<sup>4</sup> practices emerged, enabling an entire cross-functional team to focus on moving software changes to production regularly and in a sustainable and secure manner.

<sup>4</sup> DevSecOps is about integrating security practices within the DevOps process, which is a set of practices integrating software development and IT operations processes; please see glossary.

This has been a tremendous amount of progress in a short period, but some of the historical challenges of software development are still with us. They include achieving complex system qualities such as resilience and security, meeting the needs of complex groups of stakeholders, and ensuring technical coherence across a large and complex system.

## Current Challenges with Software Architecture

Now that we have an overall industry context, let us look at the state of software and enterprise architecture today. Most commercial organizations face the following challenges when defining the architectures of their products:

- Focus on technology details rather than business context.
- Perception of architects as not delivering solutions or adding value.
- Inability of architectural practices to address the increase in speed of IT delivery and to adapt to modern delivery practices such as DevOps.
- Some architects' discomfort with or lack of appropriate skills for migrating their IT environment to cloud-based platforms.

Those issues are further discussed in this section.

## **Focus on Technology Details Rather than Business Context**

Software architectures are driven by business goals, yet gaining a deep understanding of the business and its needs is not commonly done in architecture activities. At the same time, business stakeholders do not understand the complexity of our architectures. They can get frustrated when they see that they can download easily and rapidly their favorite mobile apps on their smartphones, whereas it may take several months and a lot of effort to implement a business application in the enterprise.

Architecture teams tend to spend time documenting the current state of existing systems and then create complex architecture blueprints that try to fix their perceived shortcomings. Unfortunately, these blueprints are often derived from current IT trends or fads—for example, architectures designed these days may include some combination of microservices, serverless functions, cloud computing, and big data, regardless of their applicability to the problem at hand. The root cause of this problem may be that most IT architects are more comfortable solving technology problems than they are with solving business problems.

## **Perception of Architects as Not Adding Value**

A second issue with architecture lies with the concept of enterprise architecture and enterprise architecture organizations. Broadly speaking, architects fall into three categories: application architects, solution architects, and enterprise architects. Enterprise architects may be further categorized according to the discipline they specialize in, such as application, information, security, and infrastructure. Enterprise architects are usually located in a central group, solution architect organizations may be either centralized or distributed, and application architects often sit with the developers.

Solution and application architects concentrate on delivering solutions to business problems. Application architects focus on the application dimension of the solution, whereas solution architects are concerned with every dimension of the solution, such as application, infrastructure, security, testing, and deployment. Enterprise architects attempt to create strategies, architecture plans, architecture standards, and architecture frameworks for some significant part, or perhaps all, of the enterprise. In turn, solution and

application architects are expected to abide by those strategies, plans, standards, and frameworks when working on a project. Because some enterprise architects are not close to the day-to-day business processes, the artifacts they produce may not be considered very useful by the project teams and may even be seen as hindrances.

In addition, the implication of the enterprise versus solution and application architect terminology is that enterprise architects are not perceived as focused on providing solutions and designing applications—and therefore are perceived as being potentially part of the problem. Enterprise architects can be portrayed as academic, theoretical, and out of touch with the day-to-day realities of delivering and maintaining IT systems. To be fair, we have heard the same comments about some solution and application architects as well.

## Architectural Practices May Be Too Slow

A third issue with architecture is accelerating the speed of feedback from stakeholders, not just the speed of delivery. As the pace of the business cycles increases, the **IT organization** is expected by their business partners to deliver value more rapidly. There is significant anecdotal evidence that a lot of what gets delivered, regardless of the frequency of delivery, is not particularly valuable. By measuring the results of using the software that is delivered, teams are better able to discover what is valuable and what is not. To enable rapid change, a new class of systems has emerged: the systems of engagement, which provide capabilities that support the interactions between the outside world (prospects, customers, and other companies) and the enterprise. When creating and implementing systems of this type, we are reusing much more software from the Open Source community and cloud providers than was used in the past. Therefore, in theory, it can seem that architects need to make fewer decisions than before, as many architectural decisions are implicit in the software that is being reused. However, we believe that the role of the architect is as critical in these cases as it was in more traditional software systems.<sup>5</sup> What has changed is the nature of the decisions that need to be made and the ability of today's software developers to leverage a wider variety of toolsets.

<sup>5</sup> An interesting aspect of architecting cloud solutions is a totally new way of looking at cost. The ways a system is architected in the cloud can have a significant and immediate impact on its costs.

In contrast, the planning processes associated with enterprise architecture are best suited for the older **systems of record**, which do not need to change as often as the newer **systems of engagement**. The problem we have seen is where traditional enterprise architects insist on using the same set of strategies, frameworks, and standards for every system, including the newer systems of engagement, and this attitude creates a further disconnect between the business, the software developers, and the architects.

An underlying problem that may contribute to this state of affairs is that the “architect” metaphor may not be very suitable for the design of fast-moving software systems, at least when it is based on the common person’s understanding of how an architect works. Most people think that architects work by drawing pictures and plans that they then hand to someone else who builds the building. In reality, good architects have always been deeply involved in the building process. Apprentices for Frank Lloyd Wright at Taliesin West, Arizona, were expected to go into the desert and design and build a small dwelling that they had to sleep in.<sup>6</sup> Wright himself learned a lot by using his own home and studio in Oak Park, Illinois, as a living laboratory. Similarly, in our experience, most experienced software architects also get deeply involved in the process of building their systems.

<sup>6</sup> <https://www.architecturaldigest.com/story/what-it-was-like-work-for-frank-lloyd-wright>

## Some Architects May Be Uncomfortable with Cloud Platforms

Finally, adoption of cloud platforms is increasing exponentially, and some architects are struggling to keep their skills relevant given that evolution. The immediate availability of components and tools in cloud platforms, and the ability of development teams to utilize them for rapid application development, creates additional challenges for traditional architects. They not only struggle to keep their skills up to date but also have less ability to impact the software product being developed using traditional architecture methods and tools. They may believe that architects working for cloud vendors have more control than they do over the IT architecture at the company that they work for. The increasing prevalence of Software as a Service (SaaS) solutions further accentuates this dilemma.

# Software Architecture in an (Increasingly) Agile World

Having covered some of the current challenges faced by the architect in practice, let us turn our attention to another force that has revolutionized the software development industry during the last two decades: agility.

The first agile methodology, Extreme Programming (XP), was created by Kent Beck in March 1996 when he was working at Chrysler Corporation.<sup>7</sup> The first book on agile, *Extreme Programming Explained*, was published in October 1999,<sup>8</sup> and in the following years, many other agile methodologies inspired by XP were created. Since then, adoption of those methodologies has exceeded most expectations, and agile has now spread outside of IT into business areas, being promoted by management consultants as a great management technique.

<sup>7</sup> Lee Copeland, “Extreme Programming,” *Computerworld* (December 2001). <https://www.computerworld.com/article/2585634/extreme-programming.html>. Barry Boehm’s spiral model of software development and enhancement predates XP by a decade or more.

<sup>8</sup> Kent Beck, *Extreme Programming Explained: Embrace Change* (Addison-Wesley, 1999).

How have software architecture practitioners responded to the agile tidal wave? In many cases, poorly, in our opinion.

## The Beginnings: Software Architecture and Extreme Programming

In the beginnings, agile (i.e., XP) and software architecture ignored each other. In the minds of XP practitioners, software architects were part of the red tape that they felt was counterproductive and were trying to eliminate. If a role or an activity wasn’t seen as directly contributing to the development of executable code, the XP response was to eliminate it, and software architecture was considered part of this category. In the XP approach, the architecture emerges from code building and refactoring activities, hence the term *emergent* architecture.<sup>9</sup> There is no longer any need for architects to create explicit architectural designs because “the best architectures, requirements, and design emerge from self-organizing teams.”<sup>10</sup>

<sup>9</sup> In philosophy, systems theory, science, and art, *emergence* is a process whereby larger entities, patterns, and regularities arise through interactions among smaller or simpler entities that themselves do not exhibit such properties. See Murat Erder and Pierre Pureur, *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-centric World* (Morgan Kaufmann, 2015), p. 8.

<sup>10</sup> Agile Manifesto, Principle 11. <https://www.agilemanifesto.org/principles.html>

Unfortunately, this approach does not scale well for **quality attributes**, particularly as new requirements are identified; as refactoring activities become increasingly complex, lengthy, and costly; and as systems grow in size. Our experience suggests that teams get focused on features as they are driven by their product owners to deliver ever faster, and they constantly defer **technical debt** and **technical features**, such as achieving quality attributes.

So how did most software architects react to the first agile wave? Mostly by ignoring it. They expected agile to fade away and firmly believed that the pendulum would swing back to serious “modern” iterative methodologies such as the **Rational Unified Process (RUP)** that were deferent to architects by explicitly including architecture activities.

## Where We Are: Architecture, Agility, and Continuous Delivery

As we all know, traditional software architects didn’t get their wish—agile methodologies didn’t fade away. Architects realized that agile was here to stay and that they could help agile teams, and agile teams simultaneously realized that they needed some help from the architects if they wanted their systems to meet their overall stakeholder needs, such as scalability, reliability and security. As a result, architects have started thinking about adapting their methodologies to become more aligned to agile. Books authored by architecture thought leaders such as Simon Brown<sup>11</sup> and George Fairbanks<sup>12</sup> discuss how to effectively marry architecture and agile.

<sup>11</sup> Simon Brown, *Software Architecture for Developers: Volume 2—Visualise, Document and Explore Your Software Architecture* (Leanpub, 2017).

<sup>12</sup> George Fairbanks, *Just Enough Software Architecture: A Risk-Driven Approach* (Marshall & Brainerd, 2010).

At the same time, some of the agile approaches and methodologies have started including formal architecture steps. The **Scaled Agile Framework**

**(SAFe)**, created by Dean Leffingwell, specifically includes architecture steps grouped into an **architectural runway** and leverages the concept of intentional architecture, described by Leffingwell as “an enterprise practice designed to produce and evolve robust system architectures in an agile fashion.”<sup>13</sup> However, some people have argued that SAFe may not be very agile, as they believe that it is top-down command and control with some agile terminology.

<sup>13</sup> Dean Leffingwell, “Principles of Agile Architecture: Intentional Architecture in Enterprise-Class Systems” (2008).  
[https://scalingsoftwareagility.files.wordpress.com/2008/08/principles\\_agile\\_architecture.pdf](https://scalingsoftwareagility.files.wordpress.com/2008/08/principles_agile_architecture.pdf)

Other frameworks may be better at effectively combining architecture and agility. The Large Scale Scrum (LeSS) framework<sup>14</sup> is quite agile, with some emphasis on technical excellence. It gives a different perspective on architecture and design by providing practical tips, such as “think ‘gardening’ over ‘architecting’—create a culture of living, growing design.”<sup>15</sup> The Disciplined Agile Delivery (DaD)<sup>16</sup> framework attempts to provide a more cohesive approach to agile software development and includes an “architecture owner” role as one of its primary roles. Teams who value both architecture and real agility may be able to leverage either framework effectively.

<sup>14</sup> More with LeSS, *LeSS Framework*. <https://less.works/less/framework/index.html>

<sup>15</sup> More with LeSS, *Architecture and Design*. <https://less.works/less/technical-excellence/architecture-design#ArchitectureDesign>

<sup>16</sup> Project Management Institute, *Disciplined Agile Delivery (DaD)*.  
<https://www.pmi.org/discriminated-agile/process/introduction-to-dad>

The next step in this evolution is the realization by software developers that developing software in an agile fashion isn’t enough. They also need to deliver their software quickly to test and production environments so that it can be used by real-life users. This realization is driving the adoption of continuous delivery by software development teams, and again, some software architects have been slow in responding to this trend and adapting their architecture methodologies and tools to support this approach. We clearly need a new way of looking at architecture in a continuous delivery

world, especially because the software that we are working with may be deployed to an external cloud instead of kept on premise.

## Where We May Be Going

As adoption of cloud-centric platforms such as Amazon, Google, and Microsoft, as well as SaaS solutions such as Salesforce and Workday, gather momentum, the work of the IT organization will evolve, and the role of the architect may change even more. In the future, IT work may entail a lot more configuration and integration and a lot less code development. Code development may mainly consist of creating and maintaining services that need to be integrated into an external cloud-centric platform. Will there even be a need for architects, except those working for the vendor that owns the platform, in this scenario? We believe that the answer is a clear yes, but architects' skills will need to evolve in order to cope with new software configuration and integration challenges.

## Introducing Continuous Architecture

As we have implied, there is almost certainly less value in defining up-front architecture today, but systems still must meet their challenging quality attributes; stakeholders still have complex, conflicting, and overlapping needs; there are still many design tradeoffs to understand and make; and perhaps more than ever, there are cross-cutting concerns that need to be addressed to allow a system to meet the needs of its stakeholders. These challenges are the same ones that have always preoccupied software architects; however, the way that we practice software architecture to meet them in today's environment is going to have to change. Agility and DevOps practices are fundamentally altering the way IT professionals, including software architects, are working. Software architecture may change in how it is practiced, but we believe that it is more important now than ever.

While software architecture is still an important contributor to product delivery success, we have observed that it needs to evolve to deal with an environment where systems are typically developed as a set of parallel and largely independent components (microservices). With this style of software development, the traditional single-architect approach, where a single

architect or small group of technical leads make all the key decisions, simply ends up overloading the architects and causes development to stall. This approach to software delivery needs architectural work to be performed by more people, in smaller increments, with more focus on early delivery of value, than has often been the case historically.

One way of understanding the importance of software architecture is by using an analogy based on architecture in a physical domain. In this hypothetical scenario, let's assume that we are hired to build a near replica of the iconic Hotel Del Coronado, located in Coronado, California. This hotel was featured in the famous 1959 movie *Some Like It Hot*, where it represented the Seminole Ritz in southern Florida. A wealthy fan of the movie wants to own a replica of the hotel in Florida.

Building the original hotel was not a simple process. Work started in March 1887, and the original architectural plans were constantly revised and added to during construction. The hotel was opened in February 1888 before it was fully completed, and it has been renovated and upgraded several times during its 132 years of existence. How would we approach this project?

An agile developer may want to lay bricks to start building the replica immediately. In contrast, the enterprise architect would say that, given the hotel's complex history, this is a wasteful approach. He would want, instead, to do a lot of up-front planning and create a five-year construction plan based on current building technology and practices.

Is either of these approaches the best way to go? Probably not. The goal of continuous architecture is to bridge the gap between the two methods for an overall better result (see [Figure 1.2](#)).



**Figure 1.2** Continuous Architecture context

## Continuous Architecture Definition

What is Continuous Architecture? It is an architecture approach that follows six simple principles:

*Principle 1: Architect products; evolve from projects to products.* Architecting products is more efficient than just designing point solutions to projects and focuses the team on its customers.

*Principle 2: Focus on quality attributes, not on functional requirements.* Quality attribute requirements drive the architecture.

*Principle 3: Delay design decisions until they are absolutely necessary.* Design architectures based on facts, not on guesses. There is no point in designing and implementing capabilities that may never be used—it is a waste of time and resources.

*Principle 4: Architect for change—leverage the “power of small.”* Big, monolithic, tightly coupled components are hard to change. Instead, leverage small, loosely coupled software elements.

*Principle 5: Architect for build, test, deploy, and operate.* Most architecture methodologies focus exclusively on software

building activities, but we believe that architects should be concerned about testing, deployment, and operation, too, in order to support continuous delivery.

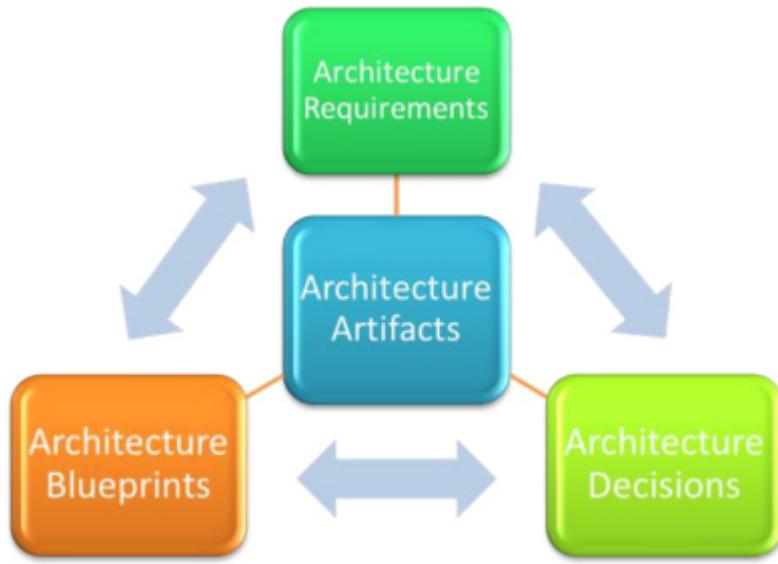
*Principle 6: Model the organization of your teams after the design of the system you are working on.* The way teams are organized drives the architecture and design of the systems they are working on.

The six principles are described in detail in *Continuous Architecture*<sup>17</sup> as well as listed in the front matter of this book. Those principles are complemented by a number of well-known architecture tools, such as utility trees and decision logs. Those tools are described in detail in *Continuous Architecture*. In this book, we also expand our practical advice by introducing a set of essential activities for architecture, which are detailed in the next chapter. The six principles, essential activities, and the tools assist the people conducting architectural activities in defining the key components of software architecture, such as

- Context of the system.
- Key functional requirements that will impact the architecture.
- Quality attributes that drive the architecture.
- Architecture and design decisions.
- Architecture blueprints.

<sup>17</sup> Erder and Pureur, *Continuous Architecture*.

Interestingly, the components of software architecture do not exist in isolation but are interrelated (see [Figure 1.3](#)). We would argue that creating a software architecture entails making a series of compromises between the requirements, the decisions, the blueprints, and even the ultimate architecture artifact—the executable code itself.



**Figure 1.3 Key components of software architecture**

For example, a decision made as a result of a performance requirement is very likely to affect a number of other quality attribute requirements, such as usability or security, and drive other decisions that will impact the architecture blueprints and eventually the capabilities being delivered by the executable code. Paraphrasing Otto von Bismarck, we could say that architecture is the art of the possible.<sup>18</sup>

<sup>18</sup> “Die Politik ist die Lehre vom Möglichen (Politics Are the Art of the Possible),” Otto Bismarck, *Fürst Bismarck: Neue Tischgespräche und Interviews*, Vol. 2 (Deutsche Verlags-Anstalt, 1899).

So how is Continuous Architecture different from other architecture approaches? To begin with, we do not think of it as a methodology but rather as a set of principles, tools, techniques, and ideas that can be thought of as an architect’s toolset to effectively deal with continuous delivery projects. There is no preset order or process to follow for using those principles, tools, techniques, and ideas, and deciding to use them is up to each architect. We have found them effective on the projects and products we have worked on, and they are dynamic and adaptable in nature. Our hope is that our reader will be inspired to adapt the contents of our Continuous Architecture toolset and extend the toolset with new ideas on how to provide architecture support to projects who want to deliver robust and effective software capabilities rapidly.

We strongly believe that leveraging our Continuous Architecture approach will help architects address and eliminate the bottlenecks that we described in the previous sections. The goal of the Continuous Architecture context is to speed up the software development and delivery process by systematically applying an architecture perspective and discipline continuously throughout the process. This enables us to create a sustainable system that delivers value to the organization over a long period.

Unlike most traditional software architecture approaches, which focus mainly on the software design and construction aspects of the software delivery life cycle (SDLC), Continuous Architecture brings an architecture perspective to the overall process, as illustrated by principle 5, *Architect for build, test, deploy, and operate*. It encourages the architect to avoid the big-architecture-up-front syndrome when software developers wait and do not produce any software while the architecture team creates complicated artifacts describing complex technical capabilities and features that may never get used. It helps the architect create flexible, adaptable, and nimble architectures that are quickly implemented into executable code that can be rapidly tested and deployed to production so that the users of the system can provide feedback, which is the ultimate validation of an architecture.

In addition, the Continuous Architecture approach focuses on delivering software rather than documentation. Unlike traditional architecture approaches, we view artifacts as a means, not as an end.

## Continuous Architecture Benefits

The cost–quality–time triangle is a well-known project management aid, which basically states the key constraints of any project (see [Figure 1.4](#)).



**Figure 1.4** Cost–quality–time triangle

The basic premise is that it is not possible to optimize all three corners of the triangle—you are asked to pick any of the two corners and sacrifice the third.

We do not claim that Continuous Architecture solves this problem, but the triangle does present a good context in which to think about benefits of Continuous Architecture. If we identify good architecture as representing quality in a software solution, then with Continuous Architecture, we have a mechanism that helps us balance time and cost. Another way of saying this is that Continuous Architecture helps us balance the time and cost constraints while not sacrificing quality.

The time dimension is a key aspect of Continuous Architecture. Architectural practices should be aligned with agile practices and should not be an obstacle. In other words, we are continuously developing and improving the architecture rather than doing it once. Continuous Architecture puts special emphasis on quality attributes (principle 2, *Focus on quality attributes, not on functional requirements*). In summary, Continuous Architecture does not solve the cost–quality–time triangle but gives us tools to balance it while maintaining quality.

Since Continuous Architecture and similar approaches were introduced during the last few years, teams leveraging these principles and tools have observed an increase of quality in the software they are delivering. Specifically, they find that they need to do much less refactoring due to architecture quality issues, such as security or performance issues. Overall

delivery timeframes are even decreasing because fewer architecture-related defects are found during software testing. This is even more noticeable in innovation projects—see [Chapter 8, “Software Architecture and Emerging Technologies.”](#) Most innovation teams focus on developing **minimum viable products** (MVPs), giving little or no thought to software architecture considerations. However, turning MVPs into actual production-grade software becomes a challenge. This may lead to a complete rewrite of the MVP before it can be used in production. Innovation teams leveraging Continuous Architecture or similar approaches tend to avoid this problem,

A missing element from the cost–quality–time triangle is sustainability. Most large enterprises have a complex technology and application landscape as a result of years of business change and IT initiatives. Agile and Continuous Delivery practices focus on delivering solutions rapidly, perhaps at the expense of addressing this complexity. Continuous Architecture tackles this complexity and strives to create a sustainable model for individual software applications as well as the overall enterprise.

Sustainability is not only about the technology but also about the people and team. Applying Continuous Architecture principles, specifically principle 6, *Model the organization of your teams after the design of the system you are working on*, enables a more cohesive team and also significantly increases the ability to sustain the knowledge of the architectural decisions supporting the software product.

Teams that apply Continuous Architecture or similar approaches at the individual application level have noticed that it enables a more sustainable delivery model and a sound technology platform that is more resilient against future change. Companies leveraging those approaches at the enterprise level have observed an increased efficiency in delivering solutions, a healthier ecosystem of common platforms, and increased knowledge sharing.

## Applying Continuous Architecture

### Continuous Architecture Provides a Set of Principles and Tools

As we said earlier, we do not aim to define a detailed architecture methodology or development process. Our main objective is to share a set of core principles and tools we have seen work in real-life practice. Applying Continuous Architecture is really about understanding the principles and ideas and applying them to the context of your environment. While doing this, you can also decide which tools you want to use as well as how to interpret the essential activities.

We have defined this approach in response to the current challenge of creating a solid architectural foundation in the world of agile and continuous delivery with a pragmatic, value-based approach. However, that does not mean that adopting continuous delivery is a prerequisite for adopting the Continuous Architecture approach. Similarly, we realize that some companies may not be ready to adapt agile methodologies everywhere. Moreover, even if a company is fully committed to agile working, there may be situations, such as working with a third-party software package, where other approaches may be more appropriate (see [Figure 1.5](#)).

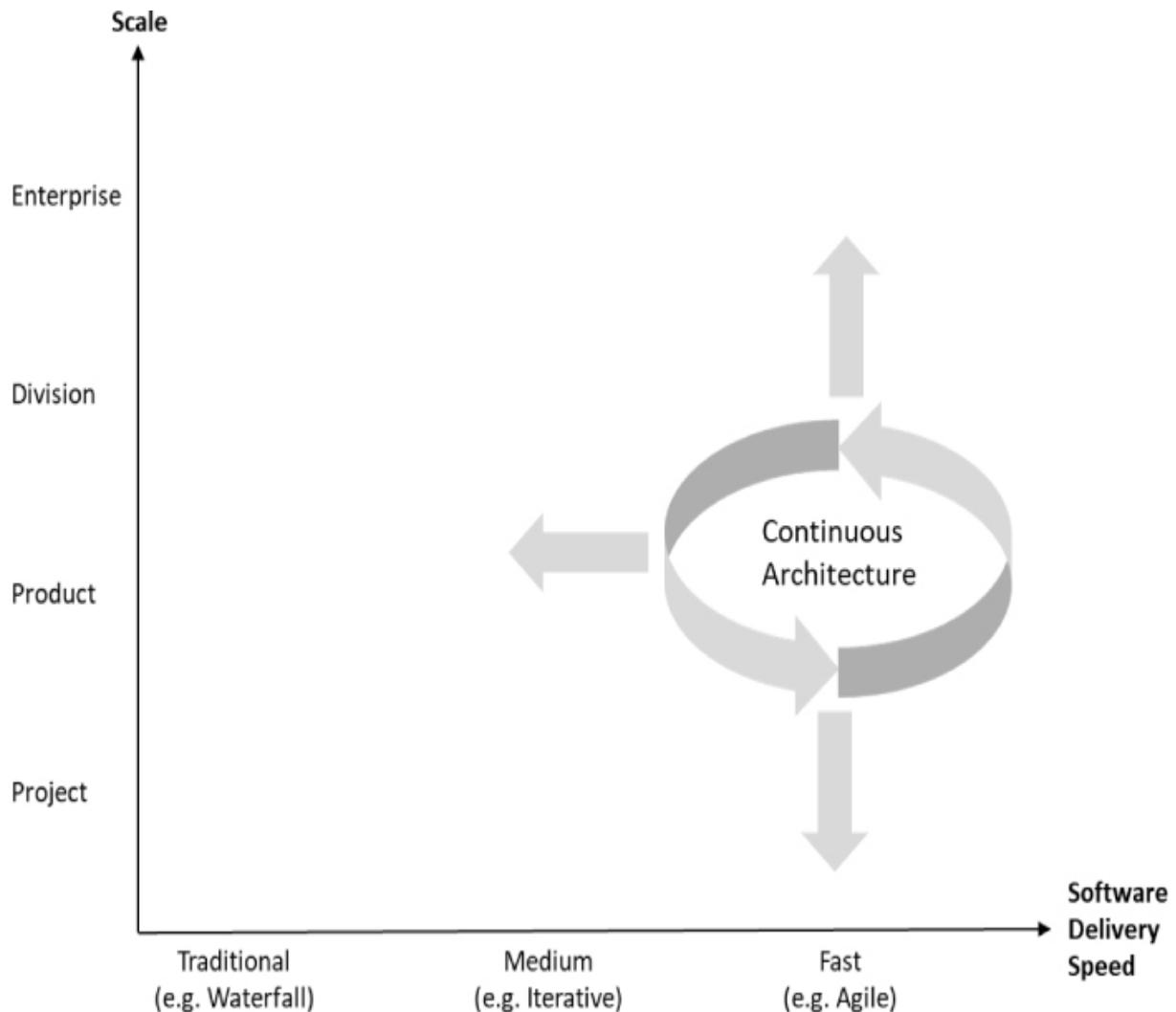


**Figure 1.5** Applying Continuous Architecture

Does this mean that Continuous Architecture would not work in this situation? Absolutely not. One of the benefits of the Continuous Architecture approach is that its tools can be easily adapted to work with other software development methodologies besides agile.

Continuous Architecture also operates in two dimensions: scale and software delivery speed (see [Figure 1.6](#)). The software delivery speed

dimension addresses how we enable architectural practices in a world of increasingly rapid delivery cycles. Although the scale dimension looks at the level at which we are operating, we believe that the Continuous Architectural principles apply consistently at all scales, but the level of focus and the tools you need to use might change.



**Figure 1.6** Continuous Architecture dimensions

In our original book, in addition to the principles and toolsets, we also focused on aspects that are relevant to the enterprise, such as governance, common services, and the role of the architect. This book attempts to provide more practical advice and operates at the scale of a software system that is based around a consistent case study.

# Introduction to the Case Study

This book is organized around a trade finance case study in order to make it as useful as possible. **Trade finance** is the term used for the financial instruments and products that are used by companies to facilitate global trade and commerce. For this case study, we will assume that we are working for a software company that is building a new trade finance application.

Our company has seen the potential to automate large portions of what is essentially a manual process and would like to create a digital platform for issuing and processing letters of credit. Initially, the aim is to offer the platform to the customers of a single financial institution, but in the longer term, the plan is to offer it as a white label (rebrandable) solution to other banks as well.

We selected this domain and case study because we believe that developing an architecture for a new trade finance platform will be a challenging task and will illuminate the obstacles that a modern architecture focus would need to overcome and the compromises that need to be made as part of this effort.

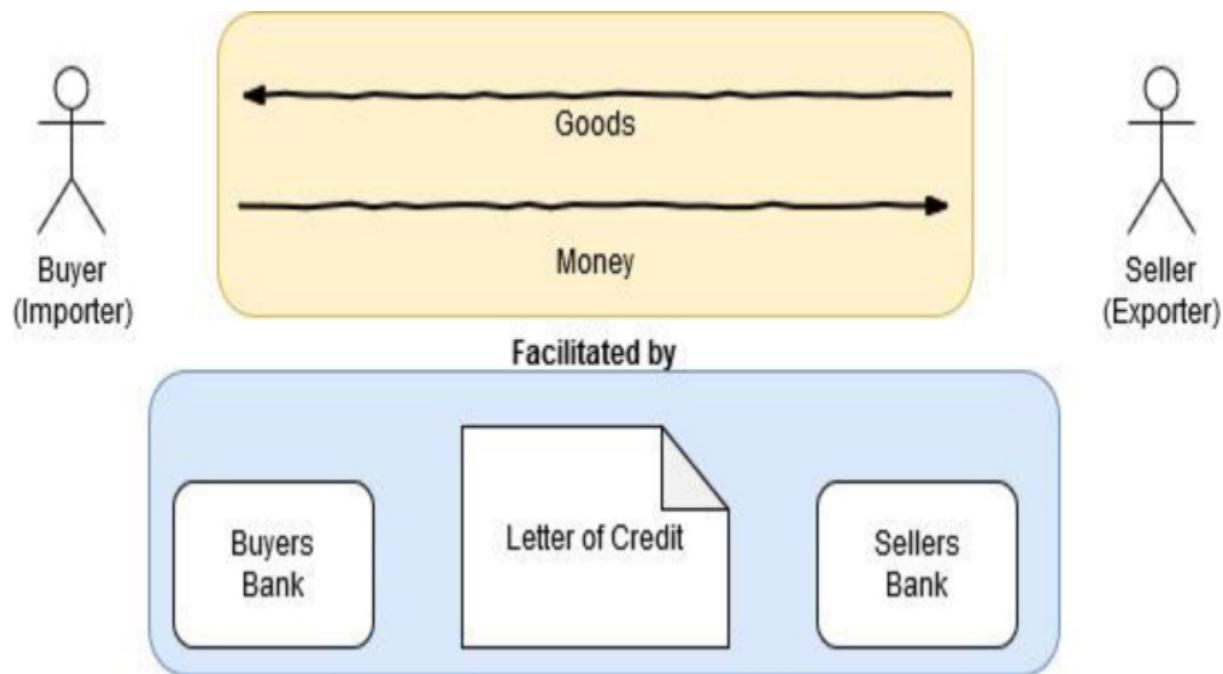
One word of caution: none of us are experts in this business domain, and therefore this case study is totally fictional. Any resemblance to real trade finance systems are purely coincidental and unintended. This is not meant to be a tutorial on how to architect and design a trade financing platform. However, we believe that having a common case study greatly enhances the ability to showcase different tradeoffs made to address quality attribute requirements. We believe that the case study provides a good balance of concerns that covers a wide area of architectural concerns. Finally, we use a trade finance case study to avoid the all-too-familiar e-commerce platform, which is used in so many books already.

We provide an overview of the case study in the next section, and a more detailed description can be found in [Appendix A](#). In the chapters, we refer to the case study and detail or expand it to illustrate how to achieve the architecturally significant requirements relevant to that chapter (e.g., additional data stores, distributed ledgers, and scalability mechanisms). However, we have not created a final version of the system. You should use

the introduction and [Appendix A](#) as a baseline from which we launch into different aspects of the Continuous Architecture approach.

## Case Study Context: Automating Trade Finance

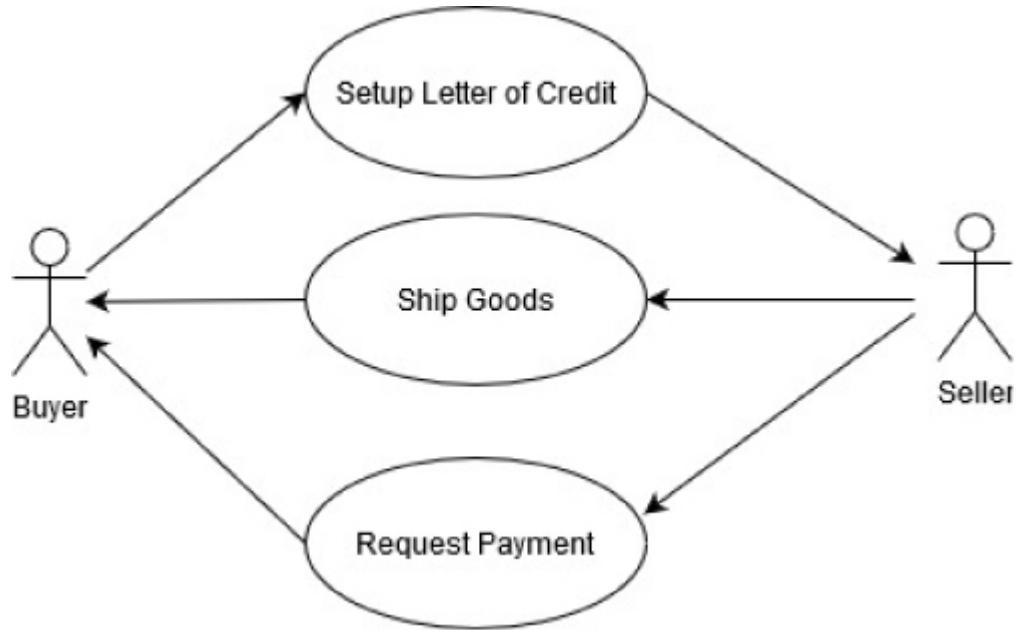
Trade finance focuses on facilitating the flow of goods and funds between buyers and sellers in global trade. The challenge trade finance tries to overcome is to ensure that payment of goods or services will be fulfilled between a buyer and a seller. There are several mechanisms for achieving this, one of which is a letter of credit (L/C). Please see [Figure 1.7](#) for an overview of L/C flows of goods and money.



**Figure 1.7** Overview of flows of goods and money for a letter of credit

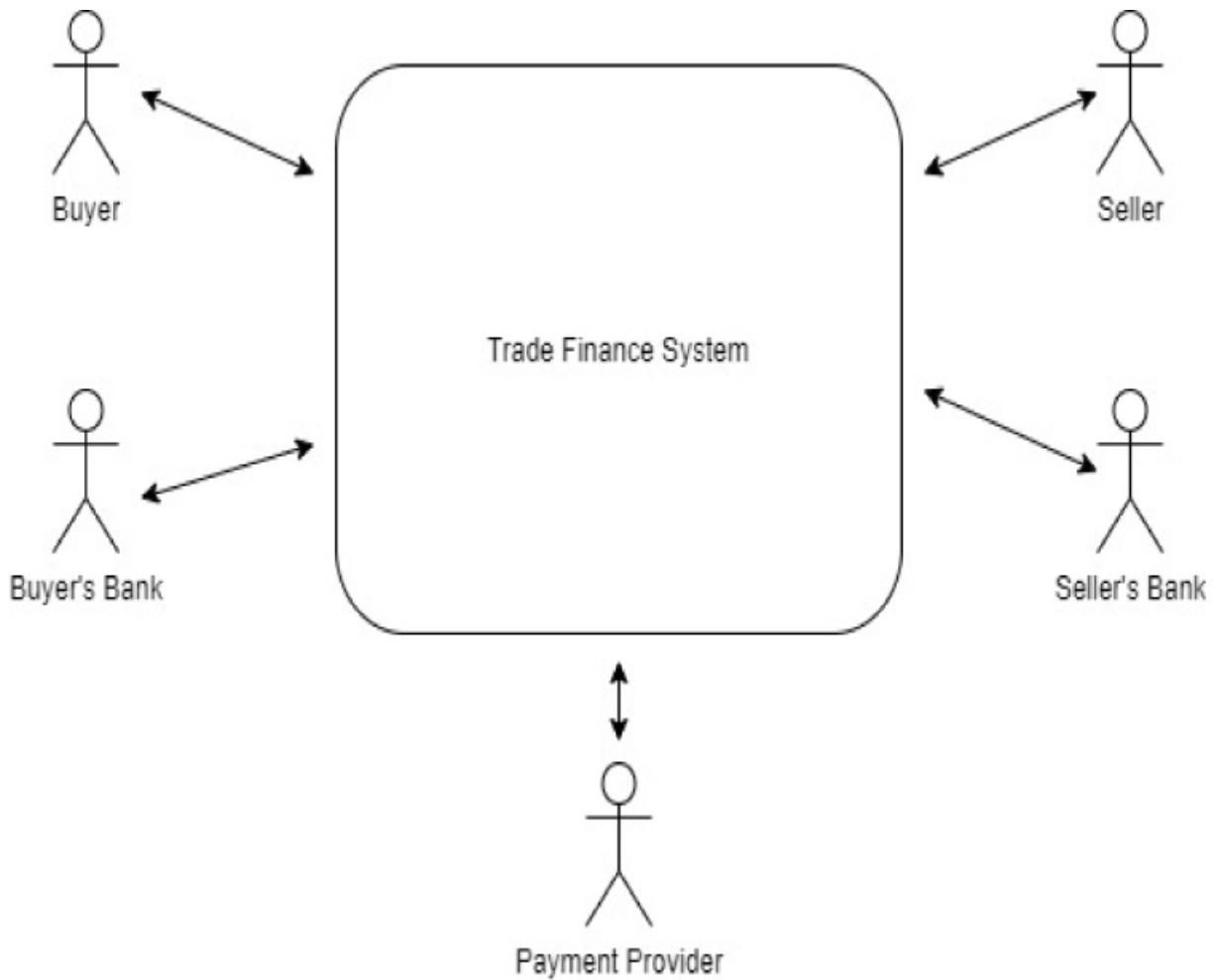
Letters of credit are standardized through the conventions established by the International Chamber of Commerce (in its UCP 600 guidelines). As well as the buyer and seller of the goods, an L/C also involves two intermediaries, typically banks, one acting for the buyer and one for the seller. The banks act as the guarantors in the transaction, ensuring that the seller is paid for its goods upon presentation of documents that prove that the goods have been shipped to the buyer.

If we ignore the intermediaries for the moment, the highest set of use cases can be depicted as shown in [Figure 1.8](#).



**Figure 1.8** *Letter of credit use cases overview*

A buyer sets up a L/C arrangement with a seller. Once that is done, the seller can ship the goods as well as request payments against the L/C. Note that, traditionally, L/Cs do not deal with tracking the actual shipment of goods but are based on provision of relevant documents during the process. If we bring the intermediaries back into the picture, we can draw the following context diagram that depicts the main actors involved in a trade finance system ([Figure 1.9](#)).



**Figure 1.9** Context diagram for a trade finance system

As mentioned previously, additional details about the case study can be found in [Appendix A](#).

## Summary

In this chapter, we explained why we believe that, although it needs to evolve, software architecture is more important now than ever. Specifically, we described what we mean by architecture. We provided an overview of the software industry today and described what we think are the challenges that software architecture is currently facing. We discussed the role of architectural practices in an increasingly agile world and the value that software architecture provides in the context of agility, DevOps, and DevSecOps.

We then explained why we believe that architecture is a key enabler in a continuous delivery and cloud-centric world and provided a definition of Continuous Architecture, described its benefits, and discussed guidelines for applying the Continuous Architecture approach. We emphasized that Continuous Architecture provides a set of principles, essential activities, and supporting tools and is not a formal methodology. Finally, we provided an introduction to and some context for the case study, which is the backbone of this book.

[Chapter 2](#), “[Architecture in Practice: Essential Activities](#),” discusses in detail the essential activities of software architecture, including focusing on architectural decisions, managing technical debt, achieving quality attributes, using architecture tactics, and implementing feedback loops.

# Chapter 2. Architecture in Practice: Essential Activities

*The architect should strive continually to simplify.*

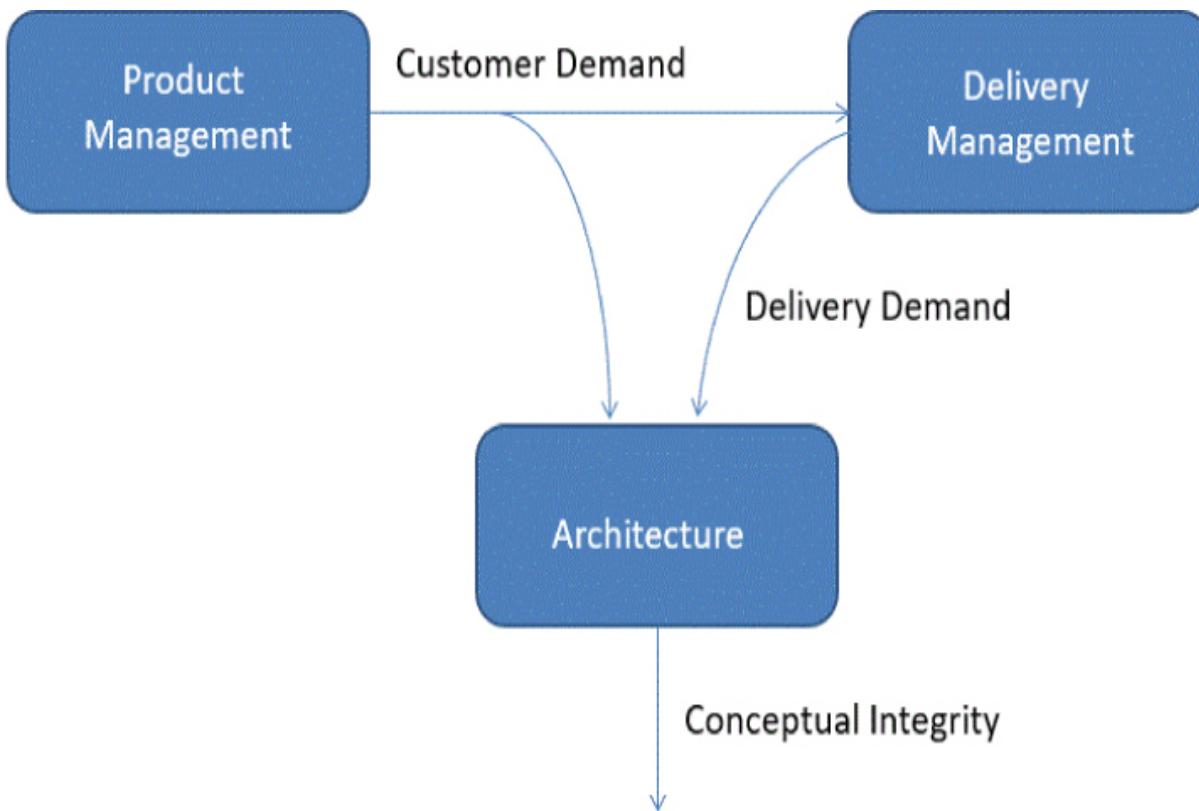
—Frank Lloyd Wright

Why is architecture important? What are the essential activities of architecture? And what practical implications do these activities have? These topics are addressed in this chapter. We already covered the definition of architecture and its relevance in [Chapter 1, “Why Software Architecture Is More Important than Ever.”](#)

To put architecture in perspective, let us focus on the development of a software system. This is an outcome of applying principle 1, *Architect products; evolve from projects to products*. For the remainder of this book, we use the term *software system* (or just *system*) to refer to the product being developed; in our case study, this is the Trade Finance eXchange (TFX) system.

As stated in our first book,<sup>1</sup> there are three key sets of activities (or roles) for any successful software system (see [Figure 2.1](#)).

<sup>1</sup> Murat Erder and Pierre Pureur, “Role of the Architect,” in *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-centric World* (Morgan Kaufmann, 2015), 187–213.



**Figure 2.1** Balancing role of architecture

Within this context, the goal of architecture is to balance customer and delivery demand to create a sustainable and coherent system. The system not only should meet its functional requirements but also should satisfy the relevant quality attributes, which we discuss later in this chapter.

A key aspect about the topic of architecture and architects is that it traditionally assumes one all-seeing and wise individual is doing architecture. In Continuous Architecture, we propose to move away from this model. We refer to “architecture work” and “architectural responsibility” instead. These terms point to the importance of the activities, while emphasizing the responsibility of the team rather than of a single person.

In his seminal book, *The Mythical Man Month*,<sup>2</sup> Frederic Brooks puts a high priority on conceptual integrity and says that having the architecture come from a single mind is necessary to achieve that integrity. We wholeheartedly agree with the importance of the conceptual integrity but believe that the same can be achieved by close collaboration in a team.

<sup>2</sup> Frederick P. Brooks Jr, *The Mythical Man-Month: Essays on Software Engineering* (Pearson Education, 1995).

Combining the Continuous Architecture principles and essential activities outlined in this section helps you protect the conceptual integrity of a software system while allowing the responsibility to be shared by the team. This should not be interpreted to mean that one individual should never undertake the role of architect, if that is appropriate for the team. What is key is that if people do undertake the role, they must be part of the team and not some external entity.

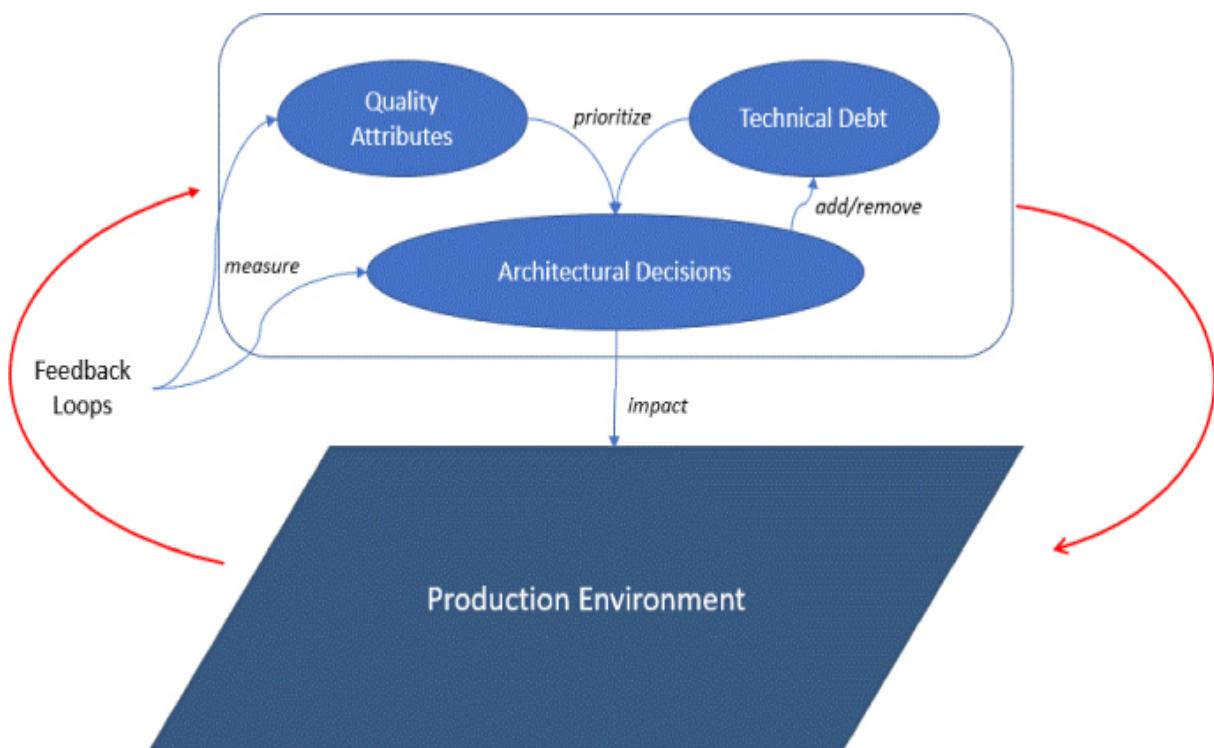
## Essential Activities Overview

From a Continuous Architecture perspective, we define the following essential activities for architecture:

- *Focus on quality attributes*, which represent the key cross-cutting requirements that a good architecture should address. Quality attributes —performance, scalability, and security, among others—are important because they drive the most significant architectural decisions that can make or break a software system. In subsequent chapters, we discuss in detail architectural tactics that help us address quality attributes.
- *Drive architectural decisions*, which are the primary unit of work of architectural activities. Continuous Architecture recommends explicitly focusing on architectural decisions because if we do not understand and capture architectural decisions, we lose the knowledge of tradeoffs made in a particular context. Without this knowledge, the team is inhibited from being able to support the long-term evolution of the software product. As we refer to our case study, we highlight key architectural decisions the team has made.
- *Know your technical debt*, the understanding and management of which is key for a sustainable architecture. Lack of awareness of technical debt will eventually result in a software product that cannot respond to new **feature** demands in a cost-effective manner. Instead, most of the team’s effort will be spent on working around the technical debt challenges—that is, paying back the debt.

- *Implement feedback loops*, which enable us to iterate through the software development life cycle and understand the impact of architectural decisions. Feedback loops are required so that the team can react quickly to developments in requirements and any unforeseen impact of architectural decisions. In today's rapid development cycles, we need to be able to course-correct as quickly as possible. Automation is a key aspect of effective feedback loops.

[Figure 2.2](#) depicts the Continuous Architecture loop that combines these elements.



**Figure 2.2** Essential activities of architecture

Clearly, the main objective of the essential activities of architecture is to influence the code running in the production environment.<sup>3</sup> As stated by Bass, Clements, and Kazman, “Every software system has a software architecture.”<sup>4</sup> The main relationships among the activities are summarized as follows:

- Architectural decisions directly impact the production environment.

- Feedback loops measure the impact of architectural decisions and how the software product is fulfilling quality attribute requirements.
- Quality attribute requirements and technical debt help us prioritize architectural decisions.
- Architectural decisions can add or remove technical debt.

<sup>3</sup> In the original *Continuous Architecture*, we refer to this as the *realized architecture*.

<sup>4</sup> Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice*, 3rd ed. (Addison-Wesley, 2012), 6. Also, according to *ISO/IEC/IEEE 42010:2011, Systems and Software Engineering –Architecture Description*, “Every system has an architecture, whether understood or not; whether recorded or conceptual.”

It might come as a surprise that we do not talk about models, perspectives, views, and other architecture artifacts. These are incredibly valuable tools that can be leveraged to describe and communicate the architecture.

However, if you do not undertake the essential activities we emphasize, architecture artifacts on their own will be insufficient. In other words, models, perspectives, views, and other architecture artifacts should be considered as a means to an end—which is to create a sustainable software system.

The following sections discuss each of the essential activities in more detail. We complete this chapter with a summary view of common themes we have observed in today’s software architectural practice that complement the essential activities.

## Architectural Decisions

If you ask software practitioners what the most visible output is from architectural activities, many will likely point to a fancy diagram that highlights the key components and their interactions. Usually, the more color and complexity, the better. The diagram is typically too difficult to read on a normal page and requires a special large-scale printer to produce. Architects want to look smart, and producing a complex diagram shows that the architect can solve extremely difficult problems! Though such diagrams give the authors and readers the false sense of being in control, they normally have limited impact on driving any architectural change. In

general, these diagrams are rarely understood in a consistent manner and provide limited insight without a voiceover from the diagram's author. In addition, diagrams are hard to change, which ends up in a divergence from the code running in the production environment that adds confusion when making architectural decisions.

This brings us to the question, What is the unit of work of an architect (or architectural work)? Is it a fancy diagram, a logical model, a running prototype? Continuous Architecture states that the unit of work of an architect is an architectural decision. As a result, one of the most important outputs of any architectural activity is the set of decisions made along the software development journey. We are always surprised that so little effort is spent in most organizations on arriving at and documenting architectural decisions in a consistent and understandable manner, though we have seen a trend in the last few years to rectify this gap. A good example is the focus on architectural decision records in GitHub.<sup>5</sup>

<sup>5</sup> <https://adr.github.io>

In our original book,<sup>6</sup> we discussed in detail what an architectural decision should look like. Following are the key points:

<sup>6</sup> Erder and Pureur, “Evolving the Architecture,” in *Continuous Architecture*, 63–101.

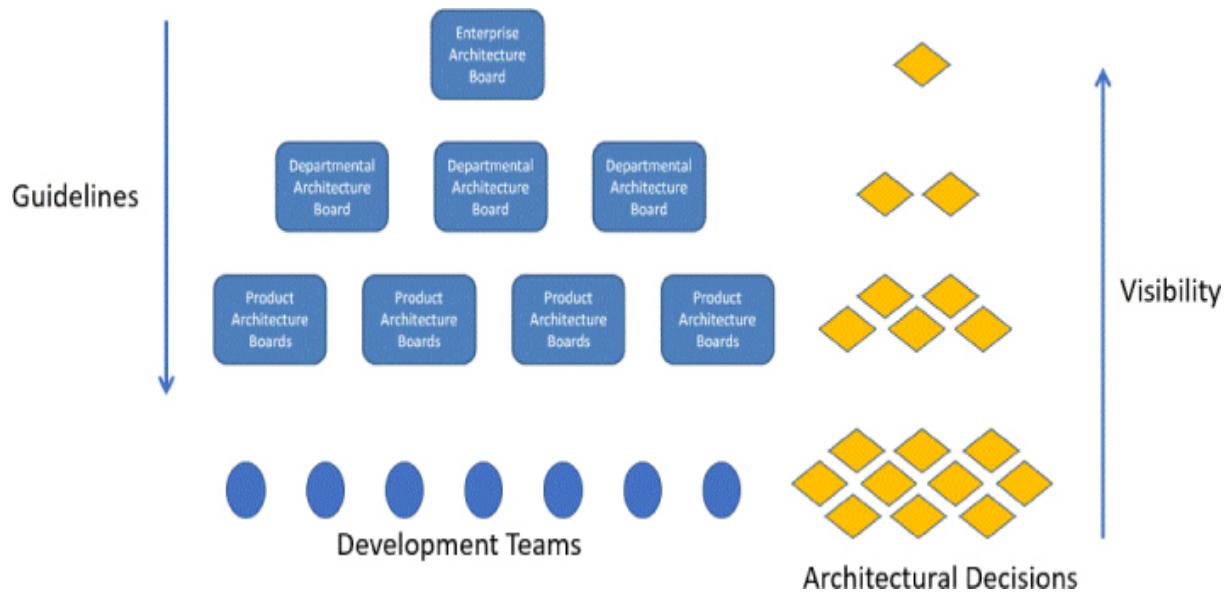
- It is important to clearly articulate all constraints related to a decision — architecture is, in essence, about finding the best (i.e., good enough) solution within the constraints given to us.
- As stated in principle 2, *Focus on quality attributes, not on functional requirements*, it is important to explicitly document quality attribute requirements.
- All options considered and rationale for coming to the decision have to be articulated.
- Tradeoff between the different options and impact on quality attributes should be considered.

Finally, this information is critical for an architectural decision: Who made this decision, and when? Appropriate accountability increases the trust in the decisions being made.

## Making and Governing Architectural Decisions

Let us look at the different types of architectural decisions in an enterprise. [Figure 2.3](#) demonstrates our recommended approach to making architectural decisions in a typical enterprise.<sup>7</sup>

<sup>7</sup> A similar view is provided by Ruth Malan and Dana Bredemeyer, “Less Is More with Minimalist Architecture,” *IT Professional* 4, no. 5 (2002): 48–47.



**Figure 2.3** Levels of architectural decisions

If we assume that an enterprise has set up governance bodies that ratify decisions, it is only natural that the higher up you go, the fewer decisions are made and the fewer reviews are conducted. For example, enterprise architecture boards make far fewer decisions than product-level governance boards. Note that the scope and significance of architectural decisions also increase with scale. However, most decisions that can impact an architecture are driven on the ground by development teams. The closer you get to implementation, the more decisions are made. Although they tend to be of a more limited scope, over time, these decisions significantly impact the overall architecture. There is nothing wrong with making more decisions at this level. The last thing we recommend is to create unnecessary burden and bureaucracy on development teams that need to be agile; they must quickly make decisions to deliver their software system. From a Continuous Architecture perspective, two elements enable us to take

advantage of aligning agile project teams to wider governance around architectural decisions:

- *Guidelines*: In reality, the probability of development teams compromising the architecture is greatly reduced if they are given clear guidelines to adhere to. For example, if there are clear guidelines around where and how to implement stored procedures, then the risk of creating a brittle architecture by writing stored procedures in random parts of the architecture can be avoided.<sup>8</sup> If you go back to [Figure 2.3](#), you see that the main job of higher governance bodies is not to make decisions but to define guidelines. The recommended approach is that there should be fewer principles the higher you go in the organization.

<sup>8</sup> It can be argued that stored procedures are more of a design decision than an architecture. A decision is a decision, and the difference between design and architecture is scale. Continuous Architecture applies at all scales.

- *Visibility*: As stated before, we do not want to stop teams from making decisions aligned with their rhythm of delivery. At the same time, we do not want the overall architecture of a system or enterprise compromised by development team decisions. To go back to our stored procedure example, we can imagine a scenario where teams put a stored procedure here and there to meet their immediate deliverables. In some cases, even the existence of these stored procedures can be forgotten, resulting in a brittle architecture that is expensive to refactor. Creating visibility of architectural decisions at all levels of the organization and sharing these decisions among different teams will greatly reduce the probability of significant architectural compromises occurring. It is not technically difficult to create visibility; all you need to do is agree on how to document an architectural decision. You can use a version of the template presented in our original book or utilize architectural decision records. You can utilize existing communication and social media channels available in the organization to share these decisions. Though technically not difficult, creating the culture for sharing architectural decisions is still difficult to realize, mainly because it requires discipline, perseverance, and open communication. There is also a natural tension between having your decisions visible to

everyone but at the same time close to the team when working (e.g., checked into their Git repository).

Let us look briefly at how the Continuous Architecture principles help us in dealing with architectural decisions. These principles are aligned with Domain-Driven Design,<sup>9</sup> which is an extremely powerful approach to software development that addresses challenges similar to those addressed by Continuous Architecture.

<sup>9</sup> [https://dddcommunity.org/learning-ddd/what\\_is\\_ddd](https://dddcommunity.org/learning-ddd/what_is_ddd)

- Applying principle 4, *Architect for change—leverage the “power of small,”* results in loosely coupled cohesive components. The architectural decisions within a component will have limited impact on other components. Some architectural decisions will still cut across components (e.g., minimally how to define the components and their integration patterns), but these decisions can also be addressed independently of component-specific decisions.
- Applying principle 6, *Model the organization of your teams after the design of the system you are working on*, results in collaborative teams that focus on delivering a set of components. This means that the knowledge sharing of relevant architectural decisions is more natural because the team is already operating in a collaborative manner.

## ***Architectural Decisions in Agile Projects***

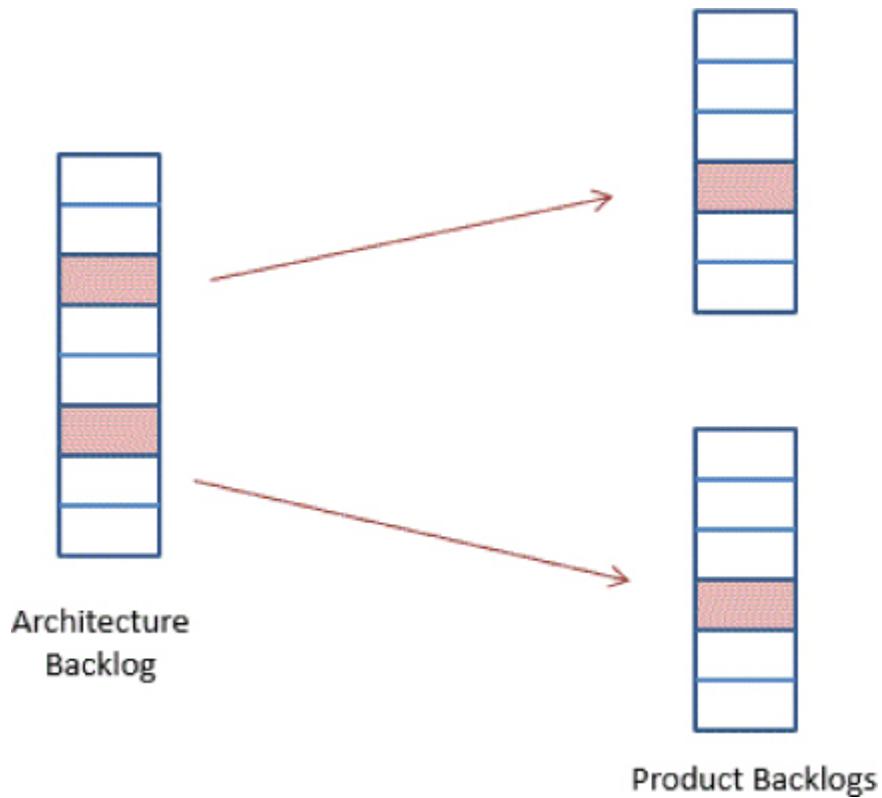
Let us now investigate architectural decisions within the context of agile development. Most technology practitioners are wary of high-level architectural direction from the ivory tower. The team will make the necessary decisions and refactor them when the need arises. We are supportive of this view. Continuous Architecture emphasizes explicitly focusing on architectural decisions rather than forgetting them in the heat of the battle: architectural decisions should be treated as a key software artifact. Making architectural decisions an explicit artifact is key for agile to scale to and link with the wider enterprise context.

By clearly defining all known architectural decisions, we are basically creating an architectural backlog. This list includes the decisions you have made and the ones you know you have to make. Obviously, the list of

architectural decisions will evolve as you make decisions and develop your product. What is important is to have a list of known architectural decisions and decide on which ones you need to address immediately. Remember principle 3, *Delay design decisions until they are absolutely necessary*.

There are two main ways in which you can integrate your architectural decisions with your product backlog. One option is to keep the architectural decision backlog separate. The second option is to have them as part of your product backlog but tagged separately. The exact approach you take will be based on what works within your context. The key point is to not lose track of these architectural decisions. [Figure 2.4](#) illustrates how the architectural decision backlog logically relates to individual product backlogs.

If you take a risk-based approach for prioritization, you will end up focusing on architecturally significant scenarios first. Then your initial set of sprints becomes focused on making key architectural decisions.



**Figure 2.4** Architectural decision and product backlogs

If you then make your architectural backlog visible to other teams and relevant architecture groups, then you have created full transparency into how you are evolving your architecture.

Although focusing on architectural decisions is an essential activity, it is still necessary to create a level of architectural description to communicate and socialize the architecture. We believe that more than 50 percent of architecture is communication and collaboration. You need such to be able to train new team members as well as explain your system to different stakeholders. Communication and collaboration are addressed in detail in the original *Continuous Architecture*.<sup>10</sup>

<sup>10</sup> Erder and Pureur, “Continuous Architecture in the Enterprise,” in *Continuous Architecture*, 215–254.

As we expand on our case study in subsequent chapters, we highlight key architectural decisions. These are examples and are not meant as a full set of decisions. In addition, we capture only some basic information for each decision, as exemplified in **Table 2.1**. For most architectural decisions, we expect that more information is captured, including constraints and detail regarding analysis and rationale.

**Table 2.1** Decision Log Entry Example

Type	Name	ID	Brief Description	Options	Rationale
Foundation al	Native Mobile Apps	FDN-1	The user interface on mobile devices will be implemented as native iOS and Android applications.	Option 1, Develop native applications.  Option 2, Implement a responsive design via a browser.	Better end-user experience. Better platform integration.  However, there is duplicated effort for the two platforms and possible inconsistency across platforms.

## Quality Attributes

For any software system, requirements fall in the following two categories:

- *Functional requirements*: These describe the business capabilities that the system must provide as well as its behavior at runtime.
- *Quality attribute (nonfunctional) requirements*: These describe the quality attributes that the system must meet in delivering functional requirements.

Quality attributes can be viewed as the *-ilities* (e.g., scalability, usability, reliability, etc.) that a software system needs to provide. Although the term *nonfunctional requirements* has widespread use in corporate software departments, the increasingly common term used in the industry is *quality attributes*. This term more specifically addresses the concern of dealing with critical attributes of a software system.<sup>11</sup>

<sup>11</sup> A more humorous criticism of nonfunctional requirements is the view that the term indicates that the requirement itself is nonfunctional.

If a system does not meet any of its quality attribute requirements, it will not function as required. Experienced technologists can point to several examples of systems that fulfill all of their functional requirements but fail because of performance or scalability challenges. Waiting for a screen to update is probably one of the most frustrating user experiences you can think of. A security breach is not an incident that any technologist would want to deal with. These examples highlight why addressing quality attributes is so critical. Quality attributes are strongly associated with architecture perspectives, where a perspective is reusable architectural advice on how to achieve a quality property.<sup>12</sup>

<sup>12</sup> Nick Rozanski and Eoin Woods, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives* (Addison-Wesley, 2012).

Formal definition of quality attributes is pretty established in the standards world, although few practitioners are aware of them. For example, the product quality model defined in ISO/IEC 25010,<sup>13</sup> part of the SQuaRE model, comprises the eight quality characteristics shown in Figure 2.5.

<sup>13</sup> International Organization for Standardization and International Electrotechnical Commission, *ISO/IEC 25010:2011 Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models* (2011). <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010>



**Figure 2.5** Product quality model

It is difficult to express quality attributes outside of a particular system context. For example, latency may be nice to have in a tax-filing application but disastrous for an autopilot. This makes it challenging to adopt such frameworks in their entirety, and defining a complete list of all quality attributes can be seen as an unnecessary academic exercise.

However, addressing the key quality attributes of your software system is one of the most important architectural considerations. The most important quality attributes need to be selected and prioritized. In practice, we can say that approximately 10 quality attribute scenarios are a manageable list for most software systems. This set is equivalent to what can be considered as architecturally significant scenarios. **Architecturally significant**<sup>14</sup> implies that the scenarios have the most impact on the architecture of the software system. These are normally driven by the quality attribute requirements that are difficult to achieve (e.g., low latency, high scalability). In addition, these scenarios are the ones that impact how the fundamental components of the system are defined, implying that changing the structure of these components in the future will be a costly and difficult exercise.

<sup>14</sup> See glossary for definition.

Experienced software practitioners know that a given set of functional capabilities can often be implemented by several different architectures with varying quality attribute capabilities. You can say that architectural

decisions are about trying to balance tradeoffs to find a good enough solution to meet your functional and quality attribute requirements.

## ***Quality Attributes and Architectural Tactics***

Functional requirements are usually well documented and carefully reviewed by the business stakeholders, whereas quality attributes are documented in a much briefer manner. They may be provided as a simple list that fits on a single page and are not usually as carefully scrutinized and tend to be truisms, such as “must be scalable” and “must be highly usable.”

However, our view is that quality attributes drive the architecture design. As stated by Bass, Clement, and Kazman, “Whether a system will be able to exhibit its desired (or required) quality attributes is substantially determined by its architecture.”<sup>15</sup> We need to make architectural decisions to satisfy quality attributes, and those decisions often are compromises, because a decision made to better implement a given quality attribute may have a negative impact on the implementation of other quality attributes.

Accurately understanding quality attribute requirements and tradeoffs is one of the most critical prerequisites to adequately architect a system.

Architectural decisions are often targeted to find the least-worst option to balance the tradeoffs between competing quality attributes.

<sup>15</sup> Bass, Clements, and Kazman, *Software Architecture in Practice*, 26.

Architectural tactics are how we address quality attributes from an architectural perspective. An architectural tactic is a decision that affects the control of one or more quality attribute responses. Tactics are often documented in catalogs in order to promote reuse of this knowledge among architects. We refer to architectural tactics throughout the book, in particular in [chapters 5 through 7](#), that focus on specific quality attributes.

## ***Working with Quality Attributes***

In the Continuous Architecture approach, our recommendation is to elicit and describe the quality attribute requirements that will be used to drive architectural decisions. But how do we describe quality attributes? A quality attribute name by itself does not provide sufficiently specific information. For example, what do we mean by *configurability*?

Configurability could refer to a requirement to adapt a system to different infrastructures—or it could refer to a totally different requirement to change the business rules of a system. Attribute names such as “availability,” “security,” and “usability” can be just as ambiguous. Attempting to document quality attribute requirements using an unstructured approach is not satisfactory, as the vocabulary used to describe the quality attributes may vary a lot depending on the perspective of the author.

A problem in many modern systems is that the quality attributes cannot be accurately predicted. Applications can grow exponentially in term of users and transactions. On the flip side, we can overengineer the application for expected volumes that might never materialize. We need to apply principle 3, *Delay design decisions until they are absolutely necessary*, to avoid overengineering. At the same time, we need to implement effective feedback loops (discussed later in this chapter) and associated measurements so that we can react quickly to changes.

We recommend leveraging the utility tree technique from the architecture tradeoff analysis method, or **ATAM**.<sup>16</sup> Documenting architecture scenarios that *illustrate* quality attribute requirements is a key aspect of this technique.

<sup>16</sup> Software Engineering Institute, *Architecture Tradeoff Analysis Method Collection*. <https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513908>

## ***Building the Quality Attributes Utility Tree***

We do not go into details of the **ATAM utility tree**, which is covered in our original book.<sup>17</sup> The most important aspect is to clearly understand the following three attributes for each scenario:

<sup>17</sup> Erder and Pureur, “Getting Started with Continuous Architecture: Requirements Management,” in *Continuous Architecture*, 39–62.

- ***Stimulus***: This portion of the architecture scenario describes what a user or any external stimulus (e.g., temporal event, external or internal failure) of the system would do to initiate the architecture scenario.
- ***Response***: This portion of the architecture scenario describes how the system should be expected to respond to the stimulus.

- *Measurement*: The final portion of the architecture scenario quantifies the response to the stimulus. The measurement does not have to be extremely precise. It can be a range as well. What is important is the ability to capture the end-user expectations and drive architectural decisions.

Another attribute you can include in defining the scenario is

- *Environment*: The context in which the stimulus occurs, including the system's state or any unusual conditions in effect. For example, is the scenario concerned with the response time under typical load or peak load?

Following is an example of a quality attribute scenario for scalability:

- *Scenario 1 Stimulus*: The volume of issuances of import letters of credit (L/Cs) increases by 10 percent every 6 months after TFX is implemented.
- *Scenario 1 Response*: TFX is able to cope with this volume increase. Response time and availability measurements do not change significantly.
- *Scenario 1 Measurement*: The cost of operating TFX in the cloud does not increase by more than 10 percent for each volume increase. Average response time does not increase by more than 5 percent overall. Availability does not decrease by more than 2 percent. Refactoring the TFX architecture is not required.

As we evolve the case study throughout this book, we provide several more examples using the same technique.

## Technical Debt

The term *technical debt* has gained a lot of traction in the software industry. It is a metaphor that addresses the challenge caused by several short-term decisions resulting in long-term challenges. It draws comparison with how financial debt works. Technical debt is not always bad—it is sometimes beneficial (e.g., quick solutions to get a product to market). The concept was first introduced by Ward Cunningham:

Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. . . . The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise.<sup>18</sup>

<sup>18</sup> Ward Cunningham, “The WyCash Portfolio Management System,” *ACM SIGPLAN OOPS Messenger* 4, no. 2 (1992): 29–30.

Although the term is used widely in the industry, it is not clearly defined. It is similar to how the term *use case* gained wide usage—but lost its original intent and clear definition. In their book *Managing Technical Debt*, Kruchten, Nord, and Ozkaya address this ambiguity and provide a comprehensive overview of the concept of technical debt and how to manage it. Their definition of technical debt is as follows:

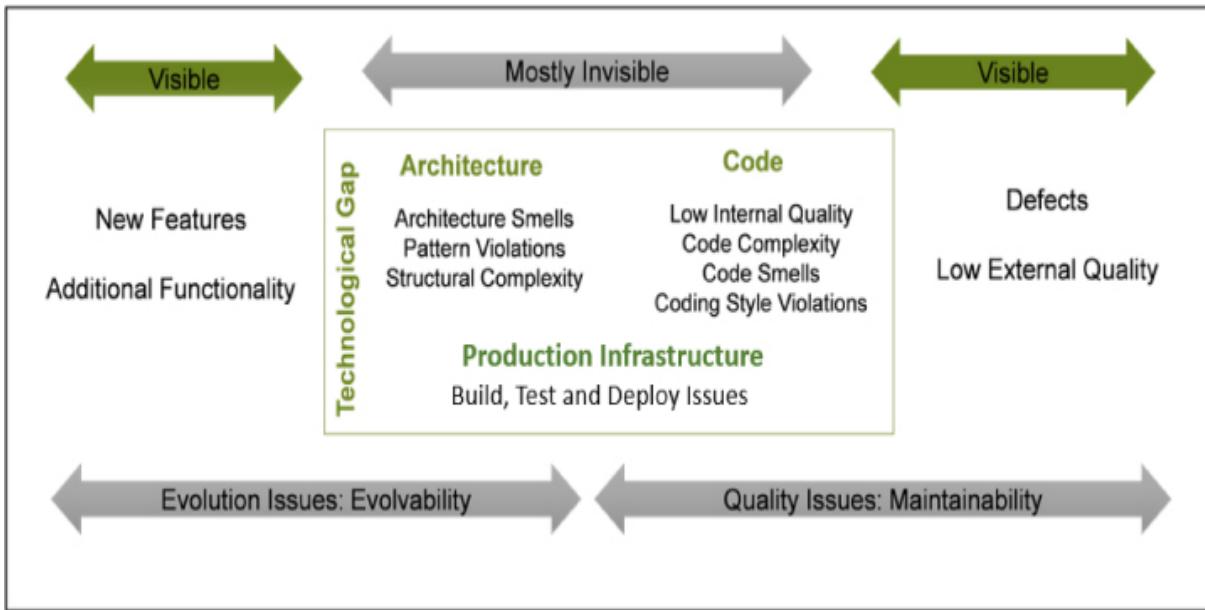
In software-intensive systems, technical debt consists of design or implementation constructs that are expedient in the short term but that set up a technical context that can make future change more costly or impossible. Technical debt is a contingent liability whose impact is limited to internal system qualities—primarily, but not only, maintainability and evolvability.<sup>19</sup>

<sup>19</sup> Philippe Kruchten, Rod Nord, and Ipek Ozkaya, *Managing Technical Debt: Reducing Friction in Software Development* (Addison-Wesley, 2019).

This is a good definition because it focuses more on the impact of technical debt and does not strictly follow the financial debt metaphor—which, though useful, is not a fully accurate way to represent the topic. The focus on maintainability and evolvability is key to how to think about technical debt. It implies that if your system is not expected to evolve, the focus on technical debt should be minimal. For example, software written for the *Voyager* spacecraft should have very limited focus on technical debt<sup>20</sup> because it is not expected to evolve and has limited maintenance opportunities.

<sup>20</sup> Or at least no intentional debt. It is almost impossible to avoid creating unintentional debt. See Kruchten, Nord and Ozkaya, *Managing Technical Debt*, principle 3, “All systems have technical debt.”

Figure 2.6 puts technical debt into context.



**Figure 2.6** Technical debt landscape. (Source: Kruchten, P., R. Nord & I. Ozkaya, Managing Technical Debt, SEI Series in Software Engineering, Addison-Wesley, 2019.)

As shown in Figure 2.6, technical debt can be divided into three categories:

- **Code:** This category includes expediently written code that is difficult to maintain and evolve (i.e., introduce new features). The Object Management Group (OMG)'s *Automated Technical Debt Measure* specification<sup>21</sup> can be used by source code analysis tools to measure this aspect. You can view the specification as standardized best practices for common topics such as managing loops, initialization of variables, and so on. Because this book is more about architecture than implementation, we do not discuss this aspect of technical debt any further.
- **Architecture:** Debt in this category is the result of architectural decisions made during the software development process. This type of technical debt is difficult to measure via tools but usually has a more significant impact on the system than other types of debt. For example, the decision to use a database technology that cannot provide the quality attributes required (e.g., using a relational database when a

basic key-value database would do) has a significant impact on the scalability and maintainability of a system.

- *Production infrastructure*: This category of technical debt deals with decisions focused on the infrastructure and code that is used to build, test, and deploy a software system. Build-test-deploy is becoming increasingly integral to software development and is the main focus of DevOps. Continuous Architecture sees the build-test-deploy environment as part of the overall architecture, as stated by principle 5, *Architect for build, test, deploy, and operate*.

<sup>21</sup> Object Management Group, *Automated Technical Debt Measure* (December 2017).  
<https://www.omg.org/spec/ATDM>

We refer readers to *Managing Technical Debt* and other books for more in-depth information on this important topic. In the next sections, we focus on recommendations of incorporating practices to identify and manage technical debt from the perspective of the architecture of a product.

---

In an interesting article, Alex Yates<sup>22</sup> proposes the term *technical debt singularity*.

<sup>22</sup> Alex Yates, “The Technical Debt Singularity,” *Observations* (2015).  
<http://workingwithdevs.com/technical-debt-singularity>

*Technology singularity* is defined as the point where computer (or artificial) intelligence will exceed the capacity of humans. After this point, all events will be unpredictable. The term was first attributed to John von Neumann:

Ever accelerating progress of technology and changes in the mode of human life, which gives the appearance of approaching some essential singularity in the history of the race beyond which human affairs, as we know them, could not continue.<sup>23</sup>

<sup>23</sup> Stanislaw Ulam, “Tribute to John von Neumann,” *Bulletin of the American Mathematical Society* 64, no. 3 (1958): 1–49.

Although the technical debt singularity does not have such dire consequences for human kind, it is still significant for impacted teams. Yates defined the technical debt singularity as follows:

So what happens if the interest we owe on our technical debt starts to exceed the number of man-hours in a working day? Well, I call that the technical debt singularity. This is the point at which software development grinds to a halt. If you spend practically all your time firefighting and you can't release often (or ever?) with confidence, I'm afraid I'm talking about you. You've pretty much reached a dead-end with little hope of making significant progress.<sup>24</sup>

<sup>24</sup> Yates, "The Technical Debt Singularity."

We can expand this to the wider enterprise and say that an enterprise has reached an architectural debt singularity when it cannot balance delivery of business demand and ongoing stability of the IT landscape in a cost-efficient manner.

---

## ***Capturing Technical Debt***

We recommend creating a technical debt registry as a key artifact for managing the architecture of a system. In terms of visibility and linkage to product backlogs, it should be managed in a similar manner to the architectural decision backlog.

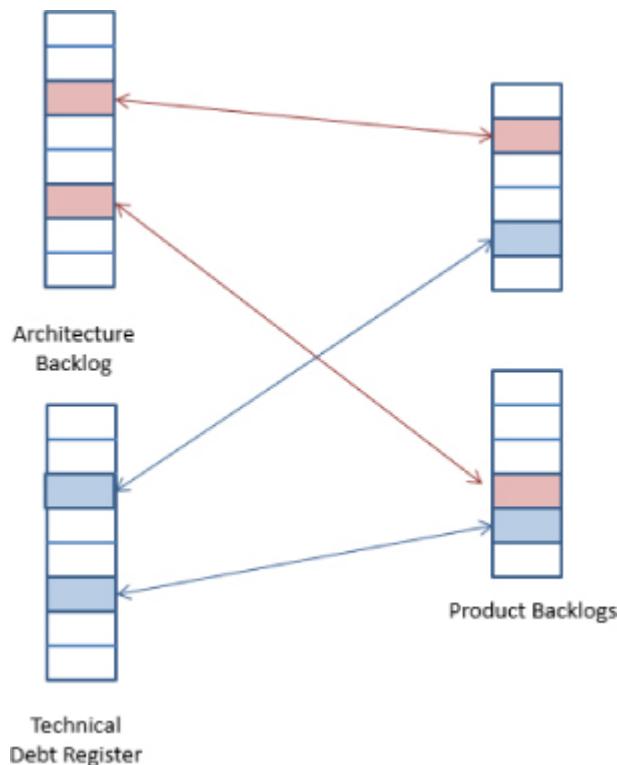
For each technical item, it is important to capture the following relevant information:

- *Consequences* of not addressing the technical debt item. The consequences can be articulated in terms of inability to meet future business requirements or limitations to the quality attributes of the product. These should be defined in a business-friendly manner because, at the end of the day, addressing technical debt will be prioritized against meeting immediate business demand.
- *Remediation approach* for addressing the technical debt item. The clearer this approach can be defined, the easier it is to make decisions on prioritization of a technical debt item against other features.

Just like the architectural decision backlog, the technical debt registry should be viewable separately. However, it does not need to be managed as a separate item.<sup>25</sup> One effective approach we have observed is to have product backlog items tagged as technical debt. When required, you can

easily pull in all technical debt items from the individual project backlogs, as shown in [Figure 2.7](#).

[25](#) See Kruchten, Nord, and Ozkaya, *Managing Technical Debt*, chapter 13, for practical advice on this topic.



**Figure 2.7** Technical debt registry and backlogs

## How to Manage Technical Debt

Once you have the technical debt registry in place, it is also important to agree on a process for the prioritization of the technical debt items. We recommend basing prioritization on the consequences of the technical debt items and not worrying too much about “technical purity.” For example, converting a file-based batch interface to an API-based real-time interface might seem like a good thing to do, but if there is limited impact on the system’s business value, it should not be prioritized.

We see two main drivers for the architectural focus on technical debt: to make appropriate architectural decisions and to influence prioritization of future releases.

While making an architectural decision, it is important to understand if we are alleviating any existing technical debt items or introducing new technical debt. This ensures that we keep the perspective of the long-term conceptual integrity of the product at each step.

Now, let us look at how prioritization of backlog items works. In an agile model, it is the product owner who decides what items should be prioritized. Even if you do not operate in a fully agile model, you still have conversations about prioritization and budget with your business stakeholders. If technical debt and its impact is not visible to the business stakeholders, it will always take a back seat to new features. Technical debt items are, by their nature, not clearly visible as features, and they have an impact predominantly on quality attributes.<sup>26</sup> This is where an architectural focus comes in.<sup>27</sup> The objective is to articulate the impact of delaying addressing technical debt items. If we delay addressing technical debt for too long, the software system can hit the technical debt singularity.

<sup>26</sup> It is obvious that significantly failing a quality attribute requirement (e.g., uptime) is very visible. However, most technical debt items are not that clear and usually affect the ability to respond to future capabilities in an efficient manner.

<sup>27</sup> See Kruchten, Nord, and Ozkaya, *Managing Technical Debt*, principle 6, “Architecture technical debt has the highest cost of ownership.”

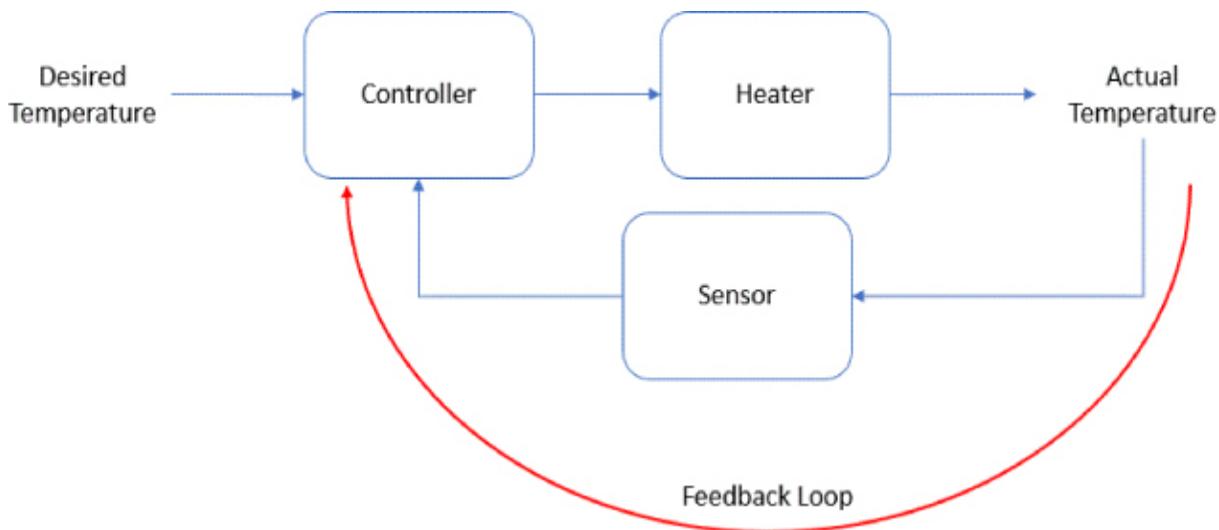
Another tactic for making sure technical debt is not lost in the rush for new features is to carve out a proportion of each release to address technical debt. How to categorize your backlog items is a wide area that is not in the scope of this book; however, a compelling view is offered by Mik Kersten.<sup>28</sup> He states that there are four types of items (i.e., flow items) to be considered in the backlog: features, defects, technical debt, and risk (e.g., security, regulatory).

<sup>28</sup> Mik Kersten, *Project to Product: How to Survive and Thrive in the Age of Digital Disruption with the Flow Framework* (IT Revolution Press, 2018).

To limit our scope to an achievable size, we decided not to discuss technical debt in the rest of this book. However, we believe that it is an important area for architects to actively manage and refer you to the references provided.

# Feedback Loops: Evolving an Architecture

Feedback loops exist in all complex systems from biological systems such as the human body to electrical control systems. The simplest way to think about feedback loops is that the output of any process is fed back as an input into the same process. An extremely simplified example is an electrical system that is used to control the temperature of a room (see [Figure 2.8](#)).



**Figure 2.8** Feedback loop example

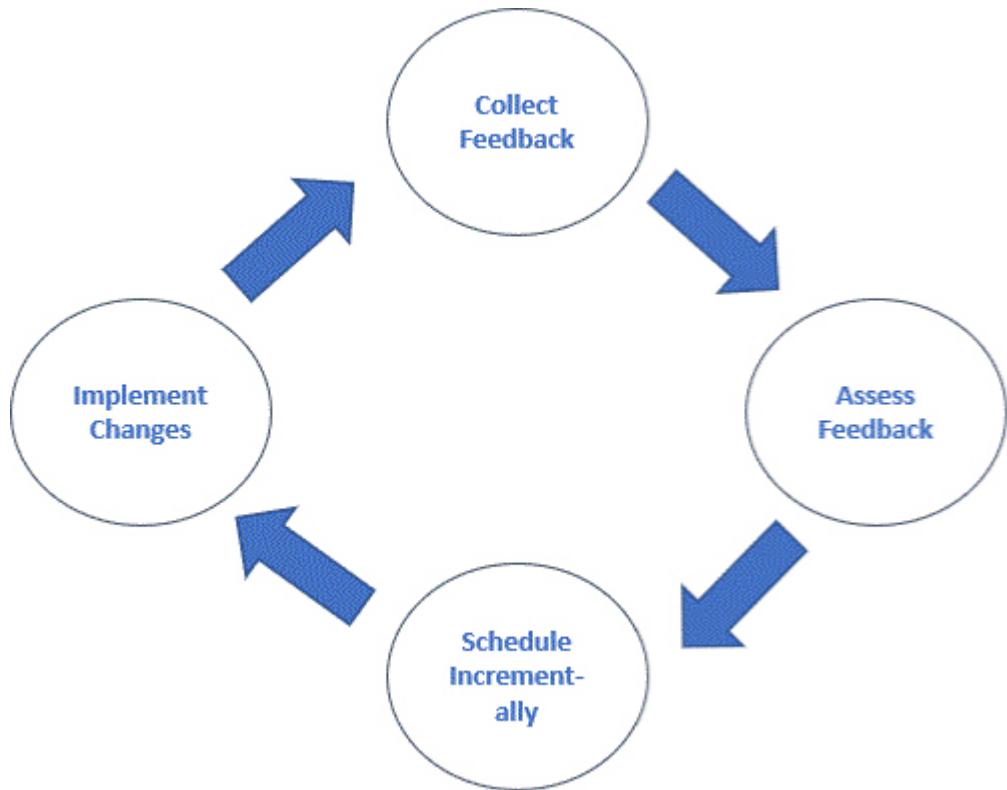
In this simple example, a sensor provides a reading of the actual temperature, which allows the system to keep the actual temperature as close as possible to the desired temperature.

Let us consider software development as a process, with the output being a system that ideally meets all functional requirements and desired quality attributes. The key goal of agile and DevOps has been to achieve greater flow of change while increasing the number of feedback loops in this process and minimizing the time between change happening and feedback being received. The ability to automate development, deployment, and testing activities is a key to this success. In Continuous Architecture, we emphasize the importance of frequent and effective feedback loops. Feedback loops are the only way that we can respond to the increasing demand to deliver software solutions in a rapid manner while addressing all quality attribute requirements.

What is a feedback loop? In simple terms, a process has a feedback loop when the results of running the process are used to improve how the process itself works in the future.

The steps of implementing a continuous feedback loop can be summarized as follows and are shown in [Figure 2.9](#):

1. *Collect measurements:* Metrics can be gathered from many sources, including fitness functions, deployment pipelines, production defects, testing results, or direct feedback from the users of the system. The key is to not start the process by implementing a complex dashboard that may take a significant amount of time and money to get up and running. The point is to collect a small number of meaningful measurements that are important for the architecture.
2. *Assess:* Form a multidisciplinary team that includes developers, operations, architects, and testers. The goal of this team is to analyze the output of the feedback—for example, why a certain quality attribute is not being addressed.
3. *Schedule incrementally:* Determine incremental changes to the architecture based on the analysis. These changes can be categorized as either defects or technical debt. Again, this step is a joint effort involving all the stakeholders.
4. *Implement changes:* Go back to step 1 (collect measurement).



**Figure 2.9** Continuous architecture feedback loop

Feedback is essential for effective software delivery. Agile processes use some of the following tools to obtain feedback:

- Pair programming
- Unit tests
- Continuous integration
- Daily Scrums
- Sprints
- Demonstrations for product owners

From an architectural perspective, the most important feedback loop we are interested in is the ability to measure the impact of architectural decisions on the production environment. Additional measurements that will help improve the software system include the following:

- Amount of technical debt being introduced/reduced over time or with each release

- Number of architectural decisions being made and their impact on quality attributes
- Adherence to existing guidelines or standards
- Interface dependencies and coupling between components

This is not an exhaustive list, and our objective is not to develop a full set of measurements and associated feedback loops. Such an exercise would end up in a generic model that would be interesting but not useful outside of a specific context. We recommend that you think about what measurement and feedback loops you want to focus on that are important in your context. It is important to remember that a feedback loop measures some output and takes action to keep the measurement in some allowable range.

As architectural activities get closer to the development life cycle and are owned by the team rather than a separate group, it is important to think about how to integrate them as much as possible into the delivery life cycle. Linking architectural decisions and technical debt into the product backlogs, as discussed earlier, is one technique. Focus on measurement and automation of architectural decisions; quality attributes is another aspect that is worthwhile to investigate.

One way to think about architectural decisions is look at every decision as an assertion about a possible solution that needs to be tested and proved valid or rejected. The quicker we can validate the architectural decision, ideally by executing tests, the more efficient we become. This activity in its own is another feedback loop. Architectural decisions that are not validated quickly are at risk of causing challenges as the system evolves.

## ***Fitness Functions***

A key challenge for architects is an effective mechanism to provide feedback loops into the development process of how the architecture is evolving to address quality attributes. In *Building Evolutionary Architectures*,<sup>29</sup> Ford, Parsons, and Kua introduced the concept of the fitness function to address this challenge. They define fitness functions as “an architectural fitness function provides an objective integrity assessment of some architectural characteristics”—where architectural characteristics are what we have defined as quality attributes of a system. These are like

the architecturally significant quality attribute scenarios discussed earlier in the chapter.

<sup>29</sup> Neal Ford, Rebecca Parsons, and Patrick Kau, *Building Evolutionary Architectures* (O'Reilly Media, 2017), 15.

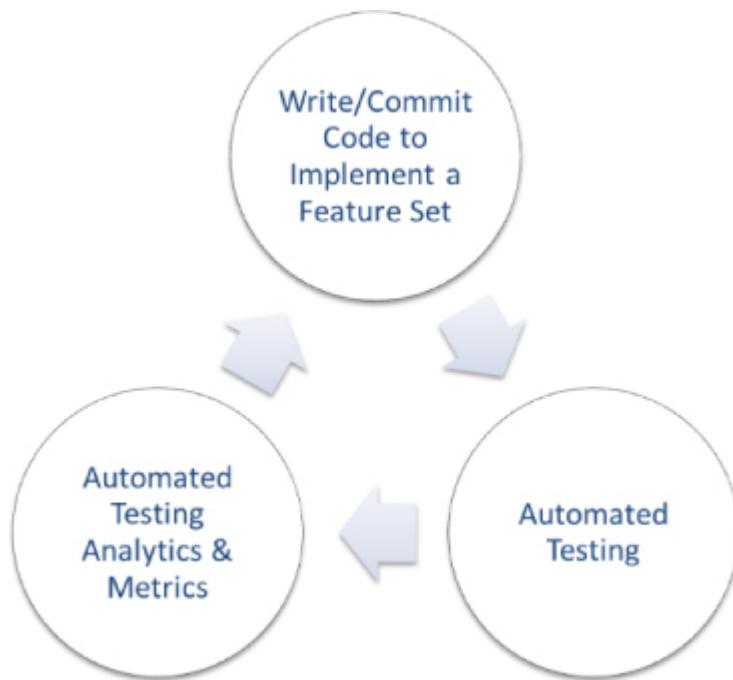
In their book, they go into detail on how to define and automate fitness functions so that a continuous feedback loop regarding the architecture can be created.

The recommendation is to define the fitness functions as early as possible. Doing so enables the team to determine the quality attributes that are relevant to the software product. Building capabilities to automate and test the fitness functions also enables the team to test out different options for the architectural decisions it needs to make.

Fitness functions are inherently interlinked with the four essential activities we have discussed. They are a powerful tool that should be visible to all stakeholders involved in the software delivery life cycle.

## ***Continuous Testing***

As previously mentioned, testing and automation are key to implementing effective feedback loops. Continuous testing implements a *shift-left* approach, which uses automated processes to significantly improve the speed of testing. This approach integrates the quality assurance and development phases. It includes a set of automated testing activities, which can be combined with analytics and metrics to provide a clear, fact-based picture of the quality attributes of the software being delivered. This process is illustrated in [Figure 2.10](#).



**Figure 2.10** Sample automated testing process

Leveraging a continuous testing approach provides project teams with feedback loops for the quality attributes of the software that they are building. It also allows them to test earlier and with greater coverage by removing testing bottlenecks, such as access to shared testing environments and having to wait for the user interface to stabilize. Some of the benefits of continuous testing include the following:

- Shifting performance testing activities to the “left” of the software development life cycle (SDLC) and integrating them into software development activities
- Integrating the testing, development, and operations teams in each step of the SDLC
- Automating quality attribute testing (e.g., for performance) as much as possible to continuously test key capabilities being delivered
- Providing business partners with early and continuous feedback on the quality attributes of a system
- Removing test environment availability bottlenecks so that those environments are continuously available

- Actively and continuously managing quality attributes across the whole delivery pipeline

Some of the challenges of continuous testing include creation and maintenance of test data sets, setup and updating of environments, time taken to run the tests, and stability of results during development.

Continuous testing relies on extensive automation of the testing and deployment processes and on ensuring that every component of the software system can be tested as soon as it is developed. For example, the following tactics<sup>30</sup> can be used by the TFX team for continuous performance testing:

<sup>30</sup> For additional details on those tactics, see Erder and Pureur, “Continuous Architecture and Continuous Delivery,” in *Continuous Architecture*, 103–129.

- Designing API-testable services and components. Services need to be fully tested independently of the other TFX software system components. The goal is to fully test each service as it is built, so that there are very few unpleasant surprises when the services are put together during the full system testing process. The key question for architects following the Continuous Architecture approach when creating a new service should be, “Can this service be easily and fully tested as a standalone unit?”
- Architecting test data for continuous testing. Having a robust and fully automated test data management solution in place is a prerequisite for continuous testing. That solution needs to be properly architected as part of the Continuous Architecture approach. An effective test data management solution needs to include several key capabilities, summarized in [Figure 2.11](#).



**Figure 2.11** *Test data management capabilities*

- Leveraging an interface-mocking approach when some of the TFX services have not been delivered yet. Using an interface-mocking tool, the TFX team can create a virtual service by analyzing its service interface definition (inbound/outbound messages) as well as its runtime behavior. Once a mock interface has been created, it can be deployed to test environments and used to test the TFX software system until the actual service becomes available.

## Common Themes in Today's Software Architecture Practice

We end this chapter with views on key trends that we see in software architectural practice. We provide only a brief overview of each topic and highlight relevant points. A detailed overview of these topics is out of the scope of this book.

### *Principles as Architecture Guidelines*

Principles are one of the most widely used type of guidelines by architecture practitioners. We define a principle as

A declarative statement made with the intention of guiding architectural design decisions in order to achieve one or more qualities of a system.<sup>31</sup>

<sup>31</sup> Eoin Woods, “Harnessing the Power of Architectural Design Principles,” *IEEE Software* 33, no. 4 (2016): 15–17.

A small set of key principles is extremely valuable if the principles are fully embraced by a team and influence the decisions they make. Principles are very valuable in communicating and negotiating decisions with key stakeholders. They allow us to have an effective dialogue highlighting future problems that can occur if the principles are violated.

For example, a team might decide to build a user interface (UI) quickly by direct access to the backend database to meet a deadline. In doing so, the team has violated the principle of “integration through APIs.” Bypassing the API will tightly couple the UI to the backend database and make future challenges in both components more difficult. Common awareness of such a principle up front will make the conversations with stakeholders much easier. They can still make the decision to go forward with the direct access route but with the understanding that they are building technical debt for their software product.

A common bad practice in the industry is to create a complete set of principles that cover all eventualities. This usually results in a long list of principles written in excruciating detail—and usually requiring lengthy editorial efforts. However, quite often, these principles end up not being embedded in the thought process of the teams that actually make the decisions.

Another challenge we have seen is how principles are written. At times, they are truisms—for example, “all software should be written in a scalable manner.” It is highly unlikely that a team would set out to develop software that is not scalable. The principles should be written in a manner that enable teams to make decisions.

As stated earlier, the most valuable principles are those that a team live and breathe while they develop a software system and make architectural decisions. They are normally a handful of basic statements.

A simple but good example for such an architectural principle is “Buy before build.” It has the following characteristics that make a good principle:

- *Clear*: Principles should be like marketing slogans—easy to understand and remember.
- *Provides guidance for decisions*: When making a decision, you can easily look to the principle for guidance. In this instance, it means that if you have a viable software product to buy, you should do that before building a solution.
- *Atomic*: The principle does not require any other context or knowledge to be understood.

## ***Team-Owned Architecture***

A key benefit of agile practices has been the focus on cross-functional and empowered teams. Effective teams can create tremendous value to an organization. It can be said that, while organizations used to look for the star developers who were multiple times more effective than an average developer, they now recognize the need for building and maintaining effective teams. This does not mean that star developers and engineers should not be acknowledged but that they are hard to find, and building effective teams in the long run is a more achievable model.

In that context, architecture activities become a team responsibility. Architecture is increasingly becoming a discipline (or skill) rather than a role. We can highlight the key skills required for conducting architectural activities as follows:<sup>32</sup>

<sup>32</sup> Eoin Woods, “Return of the Pragmatic Architect,” *IEEE Software* 31, no. 3 (2014): 10–13. <https://doi.org/10.1109/MS.2014.69><sup>69</sup>

- *Ability to design*. Architecture is a design-oriented activity. An architect might design something quite concrete, such as a network, or something less tangible, such as a process, but design is core to the activity.
- *Leadership*. Architects are not just technical experts in their areas of specialization: they’re technical leaders who shape and direct the

technical work in their spheres of influence.

- *Stakeholder focus.* Architecture is inherently about serving a wide constituency of stakeholders, balancing their needs, communicating clearly, clarifying poorly defined problems, and identifying risks and opportunities.
- *Ability to conceptualize and address systemwide concerns.* Architects are concerned about an entire system (or system of systems), not just one part of it, so they tend to focus on systemic qualities rather than on detailed functions.
- *Life cycle involvement.* An architect might be involved in all phases of a system's life cycle, not just building it. Architectural involvement often spans a system's entire life cycle, from establishing the need for the system to its eventual decommissioning and replacement.
- *Ability to balance concerns.* Finally, across all these aspects of the job, there is rarely one right answer in architecture work.

Although we state that architecture is becoming more of a skill than a role, it is still good to have a definition of the role. As mentioned earlier, in *The Mythical Man-Month*,<sup>33</sup> Brooks talks about the conceptual integrity of a software product. This is a good place to start for defining the role of architects—basically, they are accountable for the conceptual integrity of the entity that is being architected or designed.

<sup>33</sup> Brooks, *The Mythical Man-Month*.

*Continuous Architecture states that an architect is responsible for enabling the implementation of a software system by driving architectural decisions in a manner that protects the conceptual integrity of the software system.*

In our first book, *Continuous Architecture*,<sup>34</sup> we provide a detailed overview of the personality traits, skills, and communication mechanisms required for the role of an architect (or to be able to do architectural work).

<sup>34</sup> Erder and Pureur, “Role of the Architect,” in *Continuous Architecture*, 187–213.

## ***Models and Notations***

Communication is key to the success of architectural activities. Unfortunately, in the IT world, we spend a long time discussing the exact meaning of different terms (e.g., use case vs. **user story**), notation, and architectural artifacts (e.g., conceptual vs. logical vs. physical architectures).

One of the most successful attempts at creating a common notation in the software industry was the Unified Modeling Language (UML), which became an OMG standard in 1997.<sup>35</sup> In the late 1990s and 2000s, it felt as though UML was going to become the default standard for visualizing software. However, it has been waning in popularity in recent years. We are not exactly sure why this is, but one factor is that software engineering is a rapidly expanding and very young profession. As a result, most formalisms are overpowered by new technologies, fads, and ways of working. You can say that the only relevant artifact for developers is code. Any other representation requires extra effort to maintain and therefore becomes quickly outdated as a development team evolves the system.

<sup>35</sup> <https://www.omg.org/spec/UML/About-UML>

Another attempt at creating a visual language for software is ArchiMate, which was originally developed in Netherlands and became an Open Group standard in 2008.<sup>36</sup> Unlike UML, which is system focused, ArchiMate attempts to model enterprise architecture artifacts.

<sup>36</sup> <https://pubs.opengroup.org/architecture/archimate3-doc>

Although UML has gained much larger traction, there is still not an agreed-upon notation to communicate software and architecture artifacts in the industry. Paradoxically, UML and ArchiMate can make communication harder because few developers and stakeholders understand them well. Most technologists and teams normally end up drawing freeform diagrams to depict the architecture. This is a major challenge because communication is key to the success of developing and maintaining a system or enterprise architecture.

A more recent attempt at addressing this gap is the C4 model that was created by Simon Brown.<sup>37</sup> This is an interesting approach that addresses some of the challenges with more formal notations. As a philosophy, it tries

to create an approach whereby the representation of the architecture is close to the code and can be used by developers.

<sup>37</sup> <https://c4model.com>

From a Continuous Architecture perspective, we can make the following observations. As stated at the beginning of the chapter, the core elements required to drive a sustainable architecture are focus on quality attributes, architectural decisions, technical debt, and feedback loops. However, effective communication is critical: *We cannot overcommunicate!* As a result, utilizing a common language to define and communicate architectural artifacts just makes common sense. That the industry has still not found its way does not mean you should not strive for this in your area, be it a system, division, or enterprise.

Although we do not recommend a certain notation, that does not mean graphical communication and effective modeling are unimportant. Following are few key characteristics that should be considered in determining your approach:<sup>38</sup>

<sup>38</sup> <https://www.edwardtufte.com>

- *Simplicity*: Diagrams and models should be easy to understand and should convey the key messages. A common technique is to use separate diagrams to depict different concerns (logical, security, deployment, etc.).
- *Accessibility to target audience*: Each diagram has a target audience and should be able to convey the key message to them.
- *Consistency*: Shapes and connections used should have the same meaning. Having a key that identifies the meaning of each shape and color promotes consistency and enables clearer communication among teams and stakeholders.

Finally, note that for the purpose of this book, we used the UML-like notation to reflect our case study.

## ***Patterns and Styles***

In 1994, the Gang of Four—Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides—published their seminal book, *Design Patterns*.<sup>39</sup> In this book, they identified 23 patterns that address well-known challenges in object-oriented software development. Almost as important as the solutions they provided is that they introduced the concept of the design pattern and defined a manner to explain the patterns consistently.

<sup>39</sup> Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Software Architecture* (Addison-Wesley, 1995).

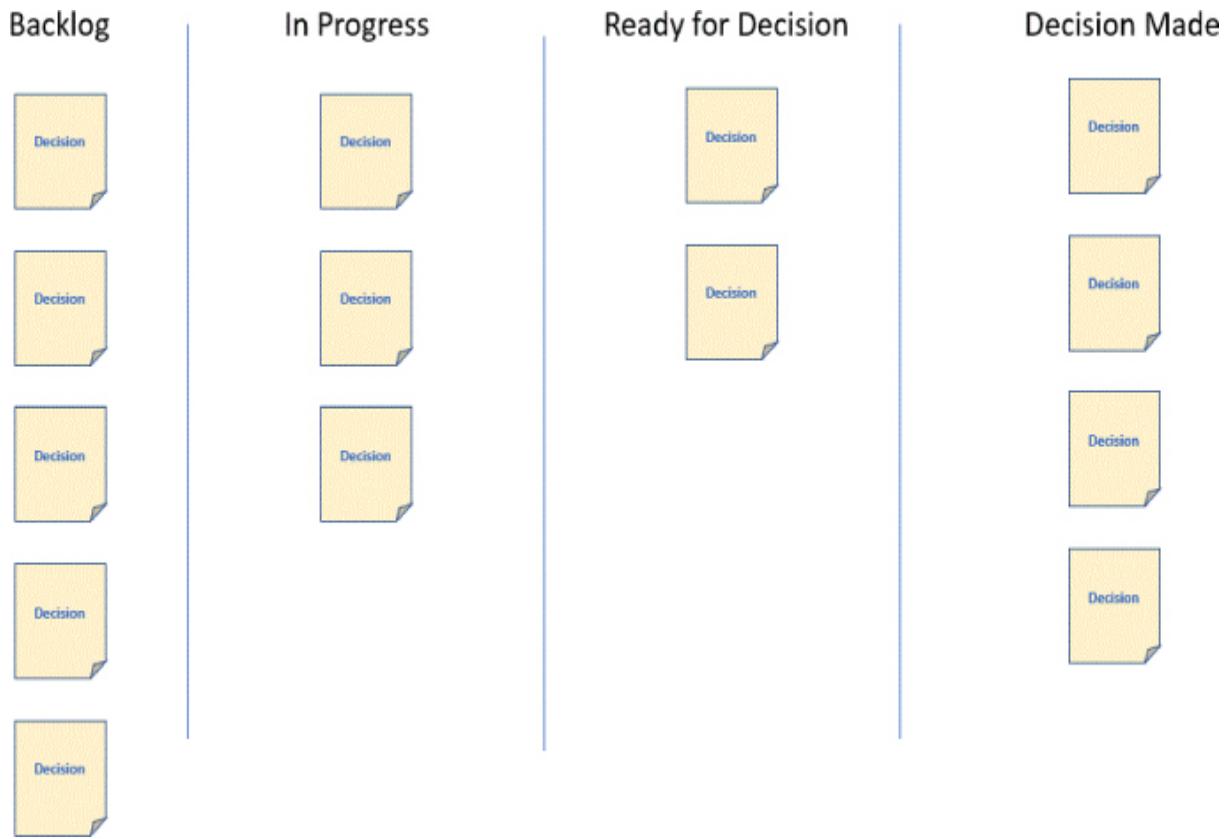
Several subsequent publications have expanded the pattern concept to different areas, from analysis to enterprise applications. More important, software designers were able to communicate with each other by referring to design patterns.

The challenge we see in the industry is that most technologists do not understand patterns or choose not to use any rigor when using them. This is particularly true when looking at tradeoffs as a result of using a pattern. Nonetheless, there is significant value in having a common pattern library within the organization. The more the patterns can be demonstrated in code or running software, the better.

## ***Architecture as a Flow of Decisions***

As mentioned, the key unit of work of architecture is an architectural decision. The topic of architectural decisions collectively defining the architecture has been prevalent in the industry for some time<sup>40</sup> and is becoming even more prominent in today's world. If you use a Kanban board to combine the view of architectural decisions as a unit of work with common software development practices focused on managing tasks, you can easily say that architecture is just a flow of decisions. [Figure 2.12](#) depicts a simple Kanban board setup that can be used to track the architectural decisions.

<sup>40</sup> Anton Jansen and Jan Bosch, “Software Architecture as a Set of Architectural Design Decisions,” in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pp. 109–120.



**Figure 2.12** Architectural decision Kanban board

This example is helpful to manage architectural decisions as a flow from an execution perspective. We recommend not only documenting architectural decisions but defining the architectural decisions you need to make up front and identifying the dependencies among them.

## Summary

This chapter discussed the essential activities of architecture and their practical implications in today's world of agile and cloud—where we are expected to deliver solutions in increasingly shorter timeframes and at increasingly larger scale. We started by defining the following essential activities that highlight why architecture is important:

- Quality attributes, which represent the key cross-cutting requirements that a good architecture should address.
- Architectural decisions, which are the unit of work of architecture.

- Technical debt, the understanding and management of which is key for a sustainable architecture.
- Feedback loops, which enable us to evolve the architecture in an agile manner.

We emphasized that the objective of the essential activities of architecture is to influence the code running in the production environment. We spoke about the importance of architecture artifacts such as models, perspectives, and views. We explained that these are incredibly valuable tools that can be leveraged to describe and communicate the architecture. However, our view is that if you do not utilize the essential activities we emphasize, they are insufficient on their own.

We then focused on each of the four essential activities, providing definitions, examples, and references to existing material in the industry. We emphasized the importance of automation and testing for feedback loops.

There is no single way of implementing these activities. We recommend adopting tools and techniques that work in your own environment and development culture.

We ended the chapter with views on select key trends that we see in the software architectural practice: principles, team-owned architecture, models and notations, patterns and styles, and architecture as a flow of decisions. For each of these trends, we provided a Continuous Architecture perspective. We believe that these trends are relevant in today's software industry and, like all, trends have benefits and pitfalls.

In the remainder of this book, we discuss a range of other aspects of software architecture and, where relevant, refer to the essential activities presented in this chapter to put their use into context.

# Chapter 3. Data Architecture

*Data is a precious thing and will last longer than the systems themselves.*

—Timothy Berners-Lee

Processing data is the reason information systems exist. You could say that the purpose of every technology that has been developed to date is to make sure that data is processed more effectively and efficiently. Because the term data is so ubiquitous, it tends to lose its meaning. This chapter explains how Continuous Architecture deals with data architecture concerns.

We start with briefly looking into what data means to help us focus on how to address the topic. We then provide an overview of key trends in the industry that make data an extremely fast-evolving and interesting architectural topic. In particular, we focus on the following trends: NoSQL and polyglot persistence, scale and availability: eventual consistency, and event sourcing and analytics. We finally dive into key architectural considerations that are required to deal with data.

The topic of data is extremely large, and we do not aim to cover all angles and nuances. Instead, we provide a sufficient overview to enable architectural thinking as it relates to data within the context of Continuous Architecture. The approach we took in identifying what topics to focus on is threefold.

First is the acknowledgment that the underlying technology available for data management is increasingly varied. There are multiple technology options and architectural styles that can be evaluated to solve a particular challenge. As described later, with these new technologies, the data architecture of a software system can no longer be seen as a separate concern. As a result, a general awareness of data technologies is a must for all technologists making architectural decisions. We provide such a technology overview, including touching on topics that directly apply to our Trade Finance eXchange (TFX) case study

Second, fundamental data-focused architectural considerations are required for any software system. They include understanding how data is managed

(data ownership and metadata), is shared (data integration), and evolves over time (schema evolution). We provide our thoughts and recommendations on these architectural considerations.

Finally, we highlight how Continuous Architecture principles apply to data architecture by demonstrating key architectural decisions that are required for the TFX case study.

We do not cover in detail data topics such as data modeling and lineage, metadata management, creating and managing identifiers, and data in motion versus data at rest, but these topics are well covered by the books listed under “[Further Reading](#)” at the end of the chapter.

## Data as an Architectural Concern

With the proliferation of connected devices and Internet usage, the volume of data in the world is increasing exponentially.<sup>1</sup> This has created a need to manage, correlate, and analyze this data in flexible ways so that we can find connections and derive insights. The technology response to this need has had a significant impact on how systems are developed. From a data architecture perspective, we can identify three main drivers:

<sup>1</sup> David Reinsel, John Gantz, and John Rydning, *Data Age 2025: The Digitization of the World from Edge to Core*, IDC White Paper Doc# US44413318 (2018), 1–29.

- Explosion of different database technologies made popular by the Internet giants known as FANG (Facebook, Amazon, Netflix, and Google).
- Ability of data science to leverage relatively cheap and scalable technology for corporations to drive strategic value from the large amount of data being generated in an increasingly digital world.
- Focus on data from a business and governance perspective. This is particularly true for highly regulated industries such as pharmaceuticals and financial services.

Before these factors came to the forefront, data architecture was the domain of a small set of data modelers, data architects, and technologists who dealt with reporting and used technologies such as data warehouses and data

marts. But in today's world of increasingly distributed systems, the focus on data needs to become central as an architectural concern.

## What Is Data?

It is an interesting exercise to ask yourself how you visualize data. Do you see a data model? Digits scrolling down a screen as in the movie *The Matrix*? A pie chart? Regardless how you visualize data, its impact on the architecture of a system cannot be overstated. Before we dive into architectural aspects, let's quickly put data in perspective by referencing the DIKW pyramid,<sup>2</sup> which is shown in Figure 3.1. It represents a commonly accepted relationship between the terms data, information, knowledge, and wisdom.

<sup>2</sup> Historically, it is thought that the pyramid originated from a few lines in a T. S. Eliot poem: “Where is the life we have lost in living? / Where is the wisdom we have lost in knowledge? / Where is the knowledge we have lost in information?” (from “The Rock,” Chorus 1).



**Figure 3.1** DIKW pyramid

Here is a simple way to interpret the DIKW pyramid:

- Data is the bits and bytes we collect.
- Information is data structured so that it can be understood and processed.
- Knowledge comes from using the data and information to answer questions and achieve goals. It is generally interpreted as answering

*how.*

- Wisdom is a little more elusive but can be seen as uncovering connections in data, information, and knowledge (i.e., focusing on *why*).

From the perspective of Continuous Architecture, how you define *data* is not that important. What we are interested in is how thinking in a data-centric manner impacts the architecture. Architectural decisions about managing data have a significant impact on the sustainability of a software system as well as on quality attributes such as security, scalability, performance, and availability.

An interesting example of how initial data architecture decisions impact the long-term evolvability of a software product is how early cable TV companies modeled their domain. Originally, cables were physically laid, and each house had only one connection. Software in this domain was architected on a house-centric model, assuming one connection and account for a house. Evolving such software products became difficult when multiple services (e.g., wireless Internet) and accounts (e.g., different occupants) started being delivered to the same address. Another example is how banks historically had account-centric models, resulting in each client having their relationship with the bank defined through the account. If the client had multiple accounts, they were seen as different relationships, making consolidation of client-level relationships difficult, especially for corporations. These then had to be transitioned to customer-centric views that enabled banks to provide multiple products and services to their clients.

## Common Language

Computer systems are unambiguous in how they deal with data, but the people who build and use the systems struggle to speak a common language. While writing this book, it took us a long time to develop a common conceptual model among us. We debated over terms such as *trader* (which we eventually ended up not using), *buyer*, *seller*, and *exporter*. Such challenges with a common vocabulary minimally create delays and frustration and, at their worst, can result in defects in the software. For example, in the trade finance domain, banks take several different roles: issuing bank, advising bank, and so on. A misunderstanding between these

different roles and how they apply to a letter of credit (L/C) could create a lot of confusion between the development team and business stakeholders.

To address this challenge, software professionals have traditionally focused on clarity of business entity definitions and their relationships (i.e., data modeling). Regardless of how you do data modeling, having a well-defined and communicated data model is a prerequisite for successful software development. Domain-Driven Design is a specific approach that has evolved out of the need to address the common language challenge.

According to Domain Language, Inc.:

Domain-Driven Design (DDD) is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts. Its premise is:

- Place the project's primary focus on the core domain and domain logic
- Base complex designs on a model
- Initiate a creative collaboration between technical and domain experts to iteratively cut ever closer to the conceptual heart of the problem.<sup>3</sup>

<sup>3</sup> [https://dddcommunity.org/learning-ddd/what\\_is\\_ddd](https://dddcommunity.org/learning-ddd/what_is_ddd)

Domain-Driven Design was introduced by Eric Evans.<sup>4</sup> One of the key concepts introduced by Domain-Driven Design is the bounded context, which states that a large domain can be split into a set of subdomains called *bounded contexts*. The details of the data model, definitions, and relationships are defined for this subdomain, and the component that is responsible for this part of the system is bound to that model. In other words, the component owns this data and is responsible for its integrity.

<sup>4</sup> Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley, 2004).

Another key concept is the ubiquitous language—basically the language that is used by the development team and business sponsors within a particular bounded context. According to Domain-Driven Design, the ubiquitous language should be reflected consistently throughout the code and any relevant design artifacts.

A key benefit of principle 6, *Model the organization of your teams after the design of the system you are working on*, is that it directs you to create cohesive teams structured around components. This principle greatly increases the ability of the teams to create a common language and is aligned with the concepts of bounded context and ubiquitous language.

We can comfortably claim that the core components of the TFX application are aligned to Domain-Driven Design principles, and each of them operates within its bounded context. However, this isn't a book on the Domain-Driven Design approach, so we suggested some titles on the topic under “[Further Reading](#)” at the end of the chapter.

The topic of common language is addressed in detail in our original book, *Continuous Architecture*.<sup>5</sup> But how does this apply to developing a system such as TFX? There are a few fundamental practices that can help in developing a common language. The first one is to use a glossary that clearly defines the terms from a business perspective. Although it can be seen as a cumbersome exercise at first, such definitions are critical for ensuring that all people have a common understanding of terms. Another interesting approach that the TFX team could have taken is to create a glossary not as a separate document but by enabling the terms to be referenced via uniform resource identifiers (URIs). These URIs can in turn be referenced in code and other artifacts.

<sup>5</sup> Murat Erder and Pierre Pureur, “Continuous Architecture in the Enterprise,” in *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-centric World* (Morgan Kaufmann, 2015).

The second practice is to take the definitions a little further and start looking at relationships among them—that is, start to define a data model. Since the objective of this model is to create a common language, it should focus on domain-level definitions and relations and not try to model it from a system-implementation perspective. The example we provided in the case study overview ([Appendix A](#)) is sufficient for people to understand the domain at a conceptual level. However, we would expect a real L/C system to have a more comprehensive model.

In summary, creating a common language is a key element for successful development of any software system. Implementing techniques from

Domain-Driven Design and practices such as referenceable glossaries can assist teams in developing a common language.

## Key Technology Trends

From the advent of the relational database and standardization of SQL in the 1980s until the explosion of Internet-scale data management in the late 2000s, managing data was relatively straightforward for most technologists. You knew you needed a database, which was almost always a relational database. And when you needed to deal with reporting at larger scales, you ended up with data marts and data warehouses. There were debates over how much to normalize data structures and star versus snowflake schemas, but these were not seen as general architectural concerns and were usually left to the data warehouse experts.

However, in the last two decades, the approach to data technology has progressed significantly, providing opportunities as well as creating challenges for technologists. We focus on the following trends:

- Demise of SQL's dominance: NoSQL and polyglot persistence
- Scale and availability: Eventual consistency
- Events versus state: Event sourcing
- Data analytics: Wisdom and knowledge from information

### Demise of SQL's Dominance: NoSQL and Polyglot Persistence

The emergence of technologies to support different data access patterns and quality attribute requirements has resulted in the ability to pick and choose the appropriate technology for the requirements.

The common term for these relatively new technologies is NoSQL—clearly indicating that we are looking at technologies that are not the traditional relational database. The core difference is that the SQL model of relational databases makes a specific set of tradeoffs to deliver atomicity, consistency, isolation, and durability (ACID)<sup>6</sup> guarantees, primarily at the expense of scalability. ACID properties are not needed by all systems. NoSQL databases improve scalability and performance by making different quality

attribute tradeoffs and by reducing the mismatch between programming language constructs and the data storage interface. Each NoSQL database is tailored for a particular type of access/query pattern. NoSQL technology is a broad topic, and there are few generalizations that hold across the whole category. However, in this chapter, we briefly touch on the different options available. There are several books and articles that provide good overviews of this topic, and the “[Further Reading](#)” section provides a recommended list.

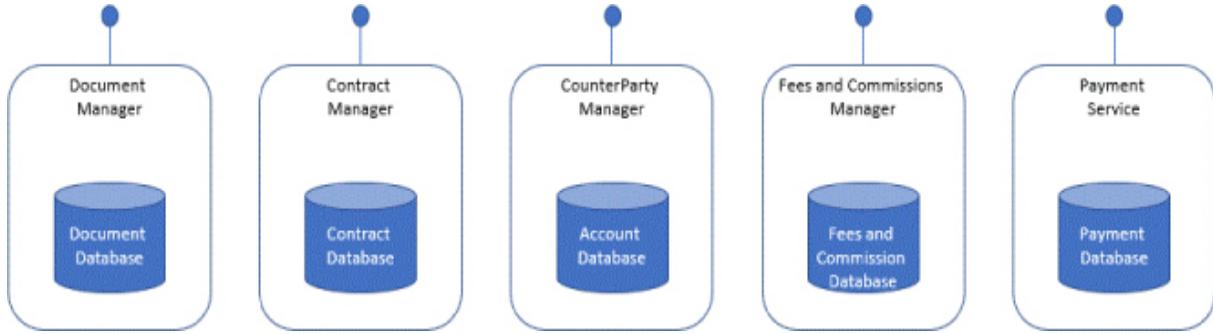
<sup>6</sup> ACID is a set of properties that databases provide to enable transaction processing.

*Polyglot persistence* means applying the right technology to address each of the different data access patterns required in an application. This approach asserts that a system can utilize more than one database platform, each suitable for the scope on which it is focused. Polyglot persistence goes hand in hand with another evolving architectural approach, [microservices](#). What this means from a data perspective is that each microservice will have its own persistence mechanism and share data with other microservices only via well-defined APIs. We touch on microservices further in [Chapter 5](#), “[Scalability as an Architectural Concern](#).”

As a side note on microservices, we want to emphasize that the overall objective is to define loosely coupled components that can evolve independently (principle 4, *Architect for change—leverage the “power of small”*). However, *micro* can be interpreted as a desire to have components with an extremely small scope, which can result in unnecessary complexity of managing dependencies and interaction patterns. In addition, it can cause performance issues. As mentioned in [Chapter 6](#), “[Performance as an Architectural Concern](#),” calls between too many microservice instances can be costly in terms of overall execution time. The decision on the scope of microservices should be driven by your context and problem domain. Think about functions (or features) that will evolve together. The goal is to have loosely coupled, internally cohesive components rather than arbitrarily small services for their own sake.

For TFX, the first major data-related decision is for each of the components to have its own database. The only allowed access to each type of data is through a well-defined API exposed by the component, as shown in [Figure 3.2](#). This decision clearly applies principle 4, *Architect for change—*

*leverage the “power of small,”* by decoupling the major components of the TFX platform.



**Figure 3.2 TFX database components**

This decision is driven mainly by the desire to loosely couple the components so that each component can be managed and released independently. It also enables independent scaling of the components. We discuss scalability in more detail in [Chapter 5](#). From a data architecture perspective, this decision allows us to select the database technology best suited for the data storage and retrieval needs of each component based on quality attribute tradeoffs. There can also be a tradeoff here between selecting the database technology that is best suited for the quality attributes and the skills of the team, as each database technology requires specific expertise.

The second important decision is what technology to use for each database. Before discussing that decision, let us provide a quick overview of different NoSQL database technologies, which can be broken down into four main categories. We do not have enough space to describe these technologies in detail. Instead, we provide a brief description and a comparison table outlining the key differences between them.

- **Key–value:** The data structure is based on a key and the value associated with the key. These are the simplest of databases, which have strong scalability and availability capabilities—but limited data model flexibility and query facilities.
- **Document:** The data structure is based on a self-describing document, usually in a format such as JavaScript Object Notation (JSON). These

databases are powerful for evolving data models because documents of the same type do not have to be of exactly the same structure.

- *Wide column*: The data structure is based on a set of columns<sup>7</sup> (versus row-based for a typical relational database). The columns also do not need to form a first normal form (1NF) table. Their benefit is scalability, but they struggle with ad hoc query patterns.

<sup>7</sup> Here we are referring to wide column stores that are part of the NoSQL family. relational databases also support column-oriented operations and there are a group of column oriented databases that follow the relational database model. See Daniel Abadi, *DBMS Musings* (2010) ([https://dbmsmusings.blogspot.com/2010/03/distinguishing-two-major-types-of\\_29.html](https://dbmsmusings.blogspot.com/2010/03/distinguishing-two-major-types-of_29.html)) for a good overview.

- *Graph*: The data structure is based on nodes and edges (i.e., entities and their relationships). These databases are very powerful for managing queries, which are predominantly based on the relationships between nodes. Unlike other NoSQL databases, their query engines are optimized for efficient graph traversal.

**Table 3.1** highlights key strengths, limitations, and typical applications for each NoSQL technology type. Note that this is a vibrant area, and each product is different. As a result, the characterizations do not apply exactly to all of the examples. **Table 3.1** is meant as a directional view on the technologies and should not be considered as a full feature comparison.

**Table 3.1** NoSQL Technology Type Comparison

Type	Main Differentiators (Strengths)	Limitations	Typical Application	Examples
Key-value	Scalability, availability, partition tolerance	Limited query functionality; cannot update part of the value separately	Storage of session data, chat room enablement, cache solutions	Memcached, Redis, Amazon Dynamo, Riak
Document	Flexible data model, easy to develop with because data representation is close to code data structures; some have ACID capabilities	Analytic processing, time series analysis; cannot update part of the document separately	Internet of Things data capture, product catalogs, content management applications; rapidly changing or unpredictable structures	MongoDB, CouchDB, Amazon Document DB
Wide column	Ability to store large datasets at high reads, scalability, availability, partition tolerance	Analytic processing, aggregation heavy workloads	Catalog searches, Time series data warehouses	Cassandra, HBase
Graph	Relationship-based graph algorithms	Transactional processing, not easy to configure for scalability	Social networking, n-degree relationships	Neo4j, Amazon Neptune, Janus Graph, GraphBase

Selection of the database technology is one of the most significant decisions that can be made for a component of a software product. Underlying technology can be very hard and expensive to change. This is particularly the case for NoSQL technologies. SQL databases provide a good separation of concerns between the database and the application, but this is less true of most NoSQL products. Implementing large-scale data solutions with NoSQL solutions results in a convergence of concerns,<sup>8</sup> where the application, data architecture, and topology become intertwined. To select the database technology, we recommend that you focus on the key quality attributes and test the quality attributes against main query patterns. If you are interested in a structured approach to evaluate different NoSQL database technologies against quality attributes, the Lightweight Evaluation

and Architecture Prototyping for Big Data (LEAP4PD) method provided by the Software Engineering Institute at Carnegie Mellon University is a good reference.<sup>9</sup>

<sup>8</sup> Ian Gorton and John Klein, “Distribution, Data, Deployment: Software Architecture Convergence in Big Data Systems,” *IEEE Software* 32, no. 3 (2014): 78–85.

<sup>9</sup> Software Engineering Institute, *LEAP(4BD): Lightweight Evaluation and Architecture Prototyping for Big Data* (2014). <https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=426058>

Coming back to the TFX product, the team decides on the following database technologies for each component (see [Table 3.2](#)).

**Table 3.2** TFX Database Technology Choices

Component	Database Technology	Rationale
Document Manager	Document database <sup>10</sup>	Supports natural structure of documents and is extensible if required
Contract Manager	Relational database	Query structure (entire document) supports expected patterns
Counterparty Manager	Relational database	Meets quality attribute requirements ACID transaction model aligns with access patterns
Fees and Commission Manager	Relational database	Skill base available in the team
Payment Service	Key-value	The actual Payment Service is external to the platform Able to meet scalability and availability requirements Basic database that will act as a log and audit trail

<sup>10</sup> The content and metadata are stored in the database. Any images, if required, can be stored directly on storage (e.g., object storage).

What [Table 3.2](#) tells us is that for the core transactional components of the system, a relational database is the selected option. As described in the table, the reasons are that the ACID transactional model aligns with the access patterns, and there are no known quality attribute concerns. The fact that SQL knowledge is available in the team and wider marketplace is also behind these decisions. However, the team has decided that two components would benefit from having different database technologies: Document Manager and Payment Service. Each has its own functional and quality attribute-related reasons for this choice. In the case of the Payment Service, we are talking about a basic database that can scale easily. In the case of the Document Store, a document database enables the team to offer a more flexible data model for different type of documents.

What is important is not the particular choices made for our TFX case study but that the service-based architectural pattern used for the TFX system enables the team to use the right technology for each service. Following principle 2, *Focus on quality attributes, not on functional requirements*, is critical in making your database technology choice.

## Scale and Availability: Eventual Consistency

Before the introduction of Internet-scale systems, data consistency and availability were not significant considerations for most systems. It was assumed that a traditional relational database would provide the required consistency for the system and would be able to meet availability requirements. Developers trusted in the relational database, including how it handled failover and replication for disaster recovery. The replication could be synchronous or asynchronous, but it was left to the database administrators (DBAs) to deal with such topics. Internet-scale systems changed all that. Single-instance relational databases could not meet the new scalability requirements, and companies had to deal with data that was globally distributed among multiple nodes. Traditional relational databases struggled to meet the nonfunctional requirements, and NoSQL databases emerged into mainstream use.

In a distributed system, making sure that the data is consistent across all the nodes while also meeting the availability requirements is difficult. This challenge is elegantly summarized in Eric Brewer's CAP theorem.<sup>[11](#)</sup> It

states that any distributed system can guarantee only two out of the following three properties: consistency, availability, and partition tolerance. Given that most distributed systems inherently deal with a partitioned architecture, it is sometimes simplified as a tradeoff between consistency and availability. If availability is prioritized, then all readers of data are not guaranteed to see the latest updates at the same time. However, at some unspecified point in time, the consistency protocol will ensure that the data is consistent across the system. This is known as *eventual consistency*, which is a key tactic in NoSQL databases for both scalability and performance.

<sup>11</sup> Eric Brewer, “CAP Twelve Years Later: How the ‘Rules’ Have Changed,” *Computer* 45, no. 2 (2012): 23–29.

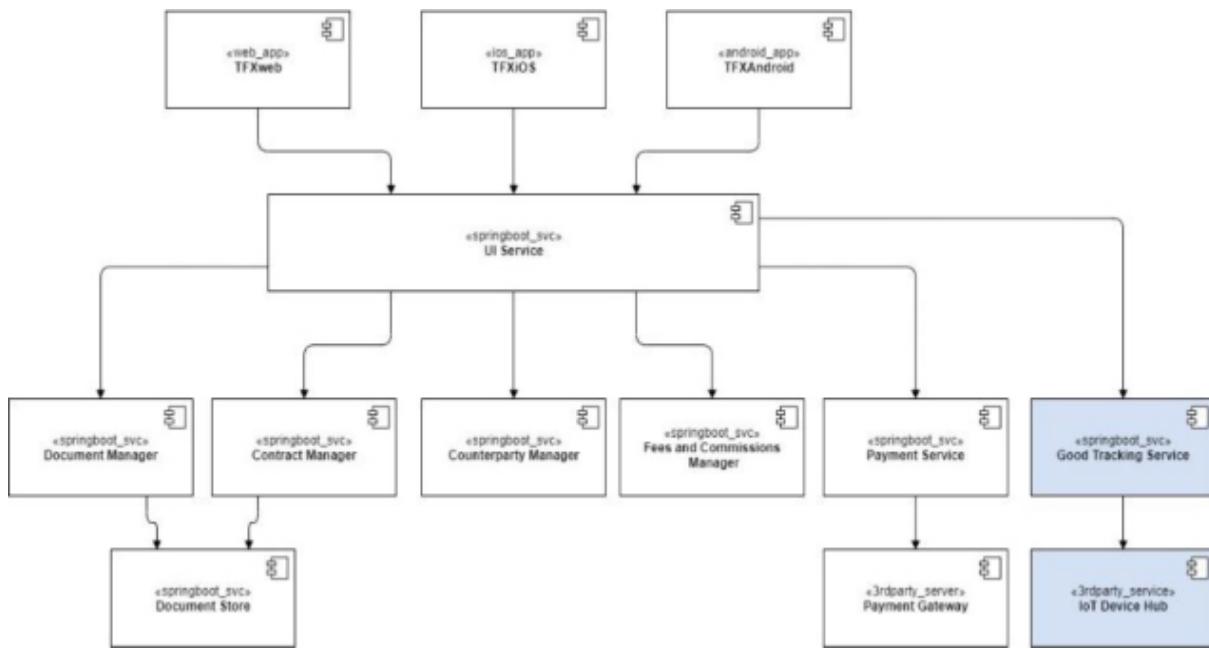
The CAP theorem is further expanded in Daniel Abadi’s PACELC,<sup>12</sup> which provides a practical interpretation of this theorem: If a partition (P) occurs, a system must trade availability (A) against consistency(C). Else (E), in the usual case of no partition, a system must trade latency (L) against consistency (C).

<sup>12</sup> David Abadi, “Consistency Tradeoffs in Modern Distributed Database System Design,” *Computer* 45, no. 2, (2012): 37–42. <http://www.cs.umd.edu/~abadi/papers/abadi-pacelc.pdf>

Looking at our TFX case study, the team has decided that for many of the components, relational database technology is sufficient, because they believe that relational databases can support the desired quality attribute requirements (performance, availability, and scalability) by applying principle 2, *Focus on quality attributes, not on functional requirements*. It is also important that their query language is SQL, which is a rich and well-established query language that most developers have at least basic knowledge of. Finally, the ACID transaction model suits the consistency requirements of the components.

However, after some time, a new requirement emerges for the TFX platform because of a change in one of the premises of the business model. Instead of relying on documentation to prove shipment of goods and trigger associated payments, the organization would like to utilize proof of actual delivery of the goods. The sales team in Asia asks to prototype a solution that integrates directly with data generated by sensors attached to the

containers that carry the goods. They partner with a company that deals with the complexity of physically installing and tracking sensors for each shipment. These sensors generate data that need to be integrated with TFX; the component that deals with this integration is called the Good Tracking Service, as shown in [Figure 3.3](#).



**Figure 3.3 Good Tracking Service**

The team now needs to decide which database technology to use for the Good Tracking Service. The key persistence requirement of the Good Tracking Service is the ability to capture all of the events that the sensors generate and make them available for the Contract Manager component.<sup>13</sup> However, there are two unknowns related to this functional requirement: future scalability requirements and the data model of the delivery sensors. It is expected that TFX will need to integrate with multiple sensor providers that will provide similar but different formats and sets of data.

<sup>13</sup> This is because events related to good tracking change the status of the contract. The integration pattern between the Good Tracking Service and Contract Manager will be via an asynchronous messaging layer.

Based on the following two reasons, the team concludes that a document store–style repository is the best option.

- A document database **schema** is flexible. Because it is expected that sensor providers will provide similar but slightly different data types, a document database gives us the ability to evolve the schema.
- A document database is very developer friendly, allowing the team to build prototypes quickly.

However, the team is conscious that scalability requirements for such an application can grow very rapidly if the business model takes off. This can result in utilizing architectural tactics for scaling databases, which is covered in [Chapter 5](#). If the team thought that scalability was a more significant driver, they could have selected key-value or wide-column database. But in the end, the flexibility and ease of use were considered as more important.

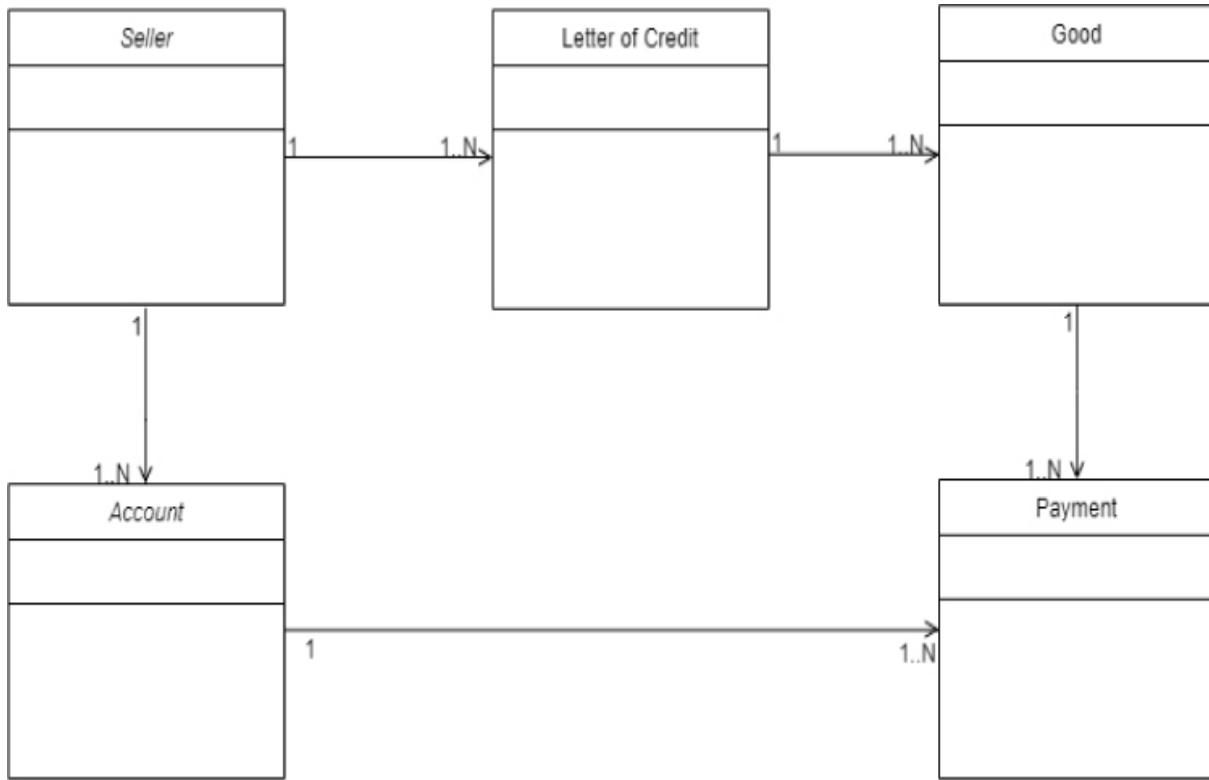
## Events versus State: Event Sourcing

Event-based systems have been around for decades but have recently become increasingly popular as an architectural style. One of the main drivers of their increased popularity is the desire to make a system as real-time as possible at large scale. We do not have sufficient space to discuss different event paradigms and architectural styles,<sup>14</sup> but we do want to provide an overview of event sourcing, which has interesting implications for how to think about events and state.

<sup>14</sup> Martin Fowler, “What Do You Mean by ‘Event-Driven’?” (2017) provides a good overview. <https://martinfowler.com/articles/201701-event-driven.html>

One common use of a traditional database system is to maintain the state of an application. By querying the database, you have a consistent view of the domain being represented. This is achieved by ensuring that the data stored in the database is consistent with the business rules of the domain using database mechanisms (e.g., triggers, referential integrity) or application code that update the database.

For example, if we had a single database for our TFX system, it would maintain a consistent state, as represented in [Figure 3.4](#).

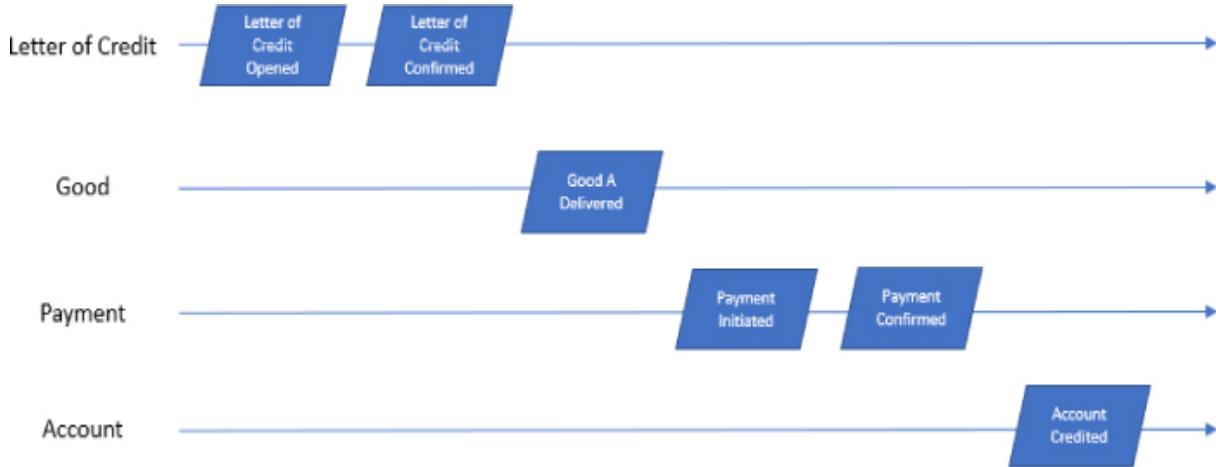


**Figure 3.4** TFX sample domain view

A relational database with the referential integrity rules implemented ensures that the state of the system at any point in time is internally consistent. For example, if there is evidence of delivery of goods through the Good Tracking Service, based on the transaction boundary we define, the database can ensure that the Good table and associated payments and account information are also updated consistently.<sup>15</sup> Another example is when database constraints ensure that the database applies all of the business consistency rules we have defined (e.g., all accounts have an associated seller as a foreign key).

<sup>15</sup> This assumes the transaction boundary in our application covers the delivery and associated payments. You would most likely define smaller transaction boundaries in a real-life system.

However, let us consider what happens if we turn our focus from maintaining an entity-based current state of the system to the view that if we capture all **events** that go into the system, we could re-create the state at any point. [Figure 3.5](#) illustrates how such an event view can be applied to the simplified TFX domain model.



**Figure 3.5 TFX domain events**

In this architectural pattern, called Event Sourcing, each entity (e.g., Account) does not track its current state in a conventional database table but instead stores all events associated with it. You can then re-create the state of the entity by replaying and reprocessing all the events.

The key benefit of this pattern is that it simplifies the writes and supports high-write workloads. The challenge is the difficulty in supporting queries. In our example, the original database can answer questions about the current state of accounts and associated payments in a straightforward way. However, a “pure” event-sourced system would need to re-create the state of the entity using the events before being able to answer the same questions. A resolution to this problem is to store the current state of the entity as well as the events or perhaps store a recent snapshot that requires only a few events to be applied to it.

This model is still significantly more complex than a conventional entity-based data model. That is why the Event Sourcing pattern is often used in conjunction with the Command Query Responsibility Segregation (CQRS) pattern.<sup>16</sup> This architectural pattern results in two different storage models within a system, one model to update information and a second model to read information. Another way to look at it is that every system request should be either a command that performs an action (resulting in a write) or a query that returns data to the caller (i.e., a read), but not both. Note that a command and an event are not the same thing. Commands are instructions to change the state of the system, whereas events are notifications.

<sup>16</sup> *Command query segregation* was first coined by Bertrand Meyer in *Object-Oriented Software Construction* (Prentice Hall, 1988). *Command query responsibility segregation* was introduced by Greg Young, *CRQS Documents*, [https://cqrss.files.wordpress.com/2010/11/cqrs\\_documents.pdf](https://cqrss.files.wordpress.com/2010/11/cqrs_documents.pdf)

For the TFX system, we have already made the decision that each component will have its own database. Each component's persistence layer decision is independent from the others. As a result, any component in the system can decide to switch from a relational database to an event store model. Event-based systems are beyond the scope of our book, and we do not explore Event Sourcing and CQRS further. However, these patterns are very useful in the design of systems that have high volumes of both update and queries.

## Data Analytics: Wisdom and Knowledge from Information

With the increased trend in analytics and artificial intelligence driven from data (which we cover in [Chapter 8, “Software Architecture and Emerging Technologies”](#)), the terms *data analytics*, *data science*, and *data engineering* have become commonplace even outside the technology world. Data analytics is an ever-expanding and interesting space from both technical and ethical perspectives. In this section, we consider only basic architectural elements related to data analytics that are relevant for our case study.

Let us briefly touch on the concepts of data science and data engineering. If we go back to the DIKW triangle, we can say that data science focuses on extracting knowledge and wisdom from the information we have. Data scientists combine tools from mathematics and statistics to analyze information to arrive at insights. Exponentially increasing amounts of data, larger compute power than ever before, and improving algorithms enable data science to offer more powerful insights each year. However, a significant challenge for the data scientist is to be able to combine data from multiples sources and convert it into information. This is where data engineering comes in. Data engineers focus on sourcing, combining, and structuring data into useful information.

As mentioned previously, the most important aspects that drive data analytics are the increasing amount of data available, the ability to utilize both unstructured and structured data, relatively cheap scalable compute

resources, and open source data middleware. The ability to capture the data first and then decide how to analyze it is one of the key benefits of modern analytics approaches. This is referred to as **schema on read versus schema on write**. The schema on read versus schema on write tradeoff can also be framed as a performance tactic. In fact, much of the motivation for using schema on read in early NoSQL databases was performance. Minimizing the amount of transformation done on the way into the system naturally increases its ability to handle more writes.

There has been a lot of hype regarding schema on read. It is a powerful architectural tactic, but you still need to understand your data. With schema on read, though, you do not have to understand and model all of it up front. In data science, significant effort is focused on data engineering (i.e., understanding and preparing your data to be able to support the analysis that is required).

Let us assume that the analytics requirements for the TFX application can be broken down into three main categories:

- *Client analytics*: Required for a particular buyer/seller or for banks and their specific information. This category has two broad patterns: simple transactional views, such as “show me the status of my account or payment,” and more detailed analytics, such as the trend of payments against a particular L/C or buyer/seller.
- *Tenant analytics*: Analytics for a particular tenant of the platform (i.e., a financial institution that utilizes the platform for its customers).
- *Cross-tenant analytics*: Analytics that can be used for anonymous **business benchmarking** by the owners and operators of the platform.

The team needs to consider multiple architectural decisions to meet these requirements. Let us walk through one way the team can address them.

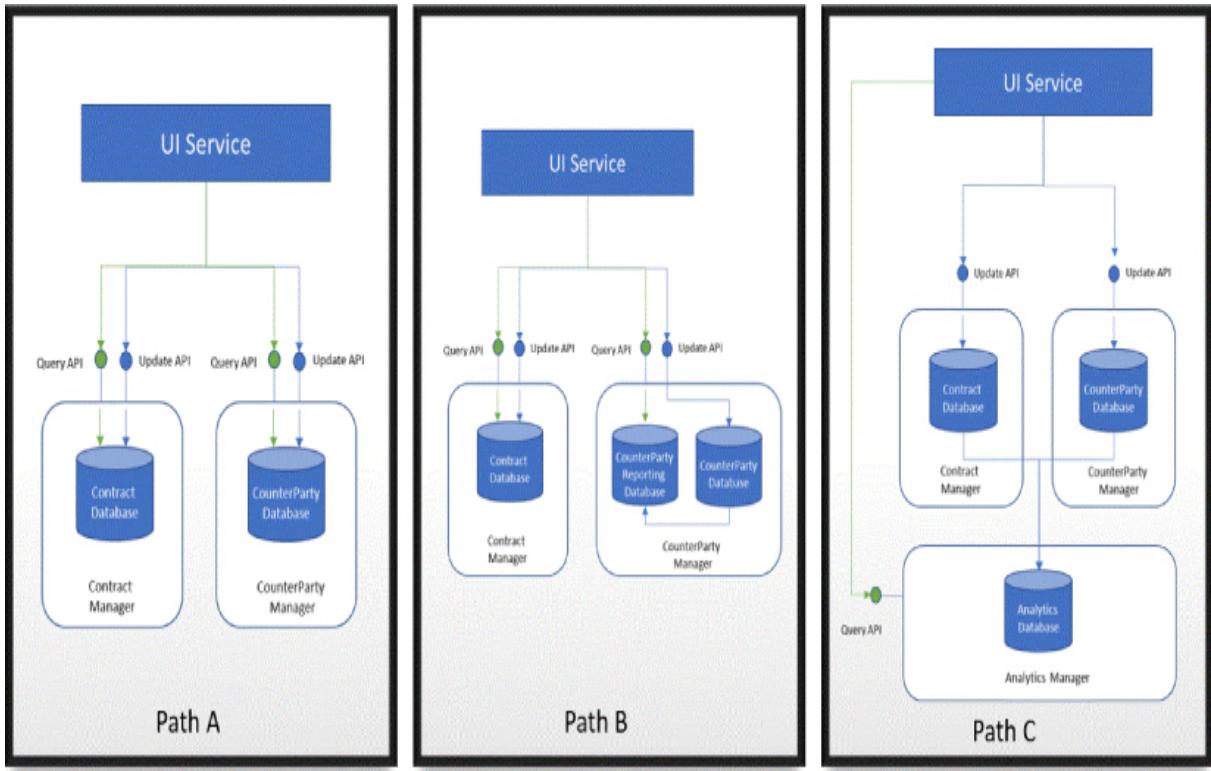
The team starts with client analytics. Remember that the data required for analytics is spread across multiple components. For illustrative purposes, we focus on two components: Counterparty Manager and Contract Manager. The first decision they need to make is whether they should create a separate analytics component. Following principle 3, *Delay design decisions until they are absolutely necessary*, they decide to see if they can

meet the requirements without adding any additional components to the system.

The team notices that a majority of the transactional queries are against the Counterparty Manager. This is expected because these queries relate to the status of accounts. The second set of transactional queries are around L/C status, which is the responsibility of the Contract Manager. The team builds a test harness for the transactional queries running against the components. The team also separates the Query API from the Update API, using an approach similar to the CQRS pattern. Based on the results of these performance tests, they can take several different paths. Note that the objective of all the options is to provide the same Query API capabilities. The UI Service can then leverage standard technology<sup>17</sup> to enable analytics capabilities, as shown in [Figure 3.6](#):

<sup>17</sup> For example, Tableau or Amazon QuickSight.

- If performance is acceptable, they make no change to the architecture —path A.
- If performance challenges are within one component (e.g., Counterparty Manager), they can go for creating a separate analytics database within that component—path B.
- If performance challenges are present in a number of components, they can create a totally separate analytics component—path C.



**Figure 3.6** TFX client analytics paths

Let us briefly look at the database technology choices for these options. It is clear that for path A, the team is not changing the database technology. Similarly, for path B, the most straightforward choice would be for the analytics database to be of the same technology type as the core database, mainly because of the ability to leverage tools provided by the database technology for replication. For the Counterparty Manager, this means a second SQL database. However, path C creates a totally separate analytics component and presents a more viable case to use different database technologies for analytics.

If the team follows path C, they come across another architectural decision: how to move the data from the core components and the analytics manager. The team considers two options:

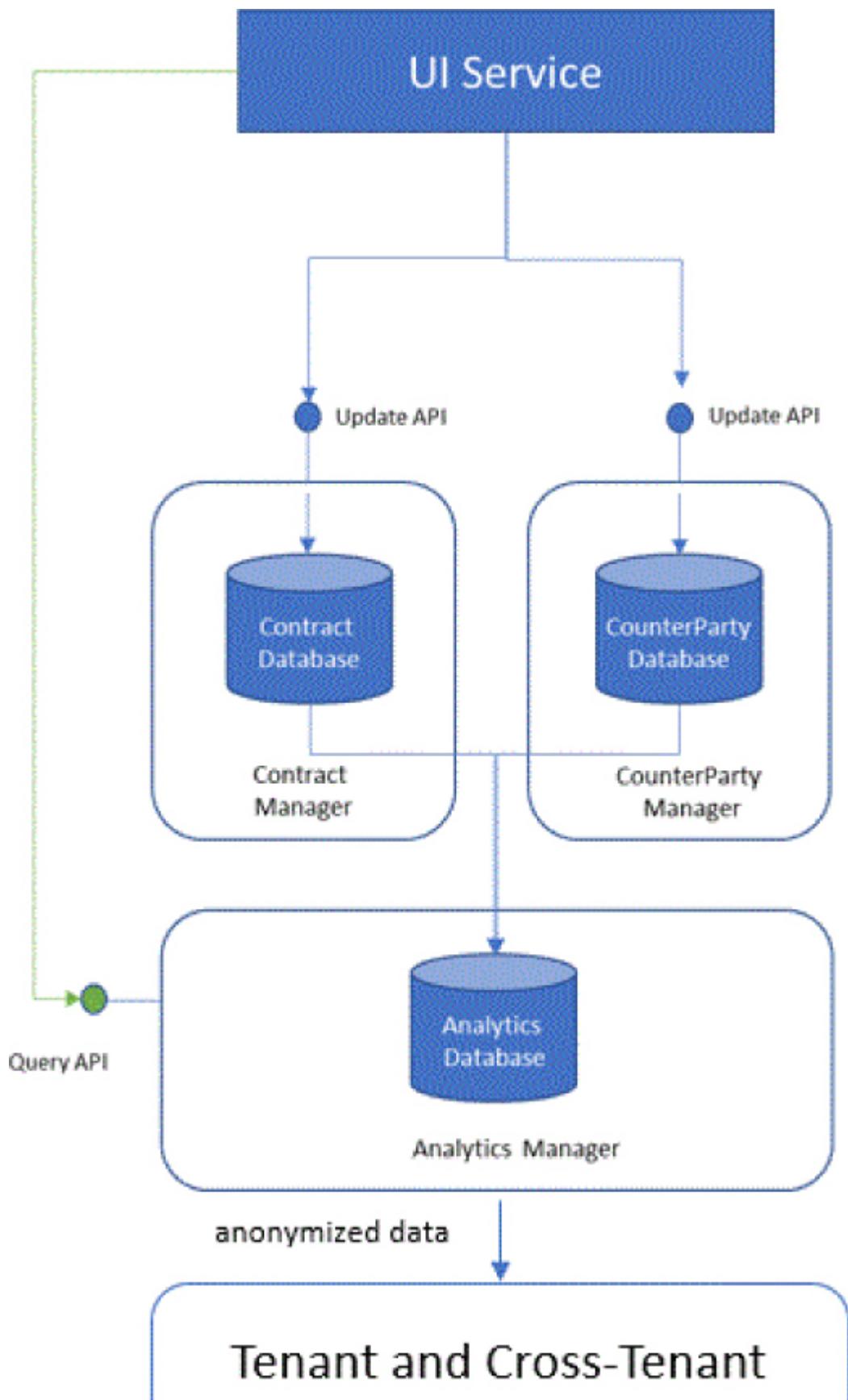
- *Decoupled (e.g., messaging):* The analytics components listen to all events that happen in the core transactional layer. This option has the benefits of fully decoupling the schema between the environments and of being a real-time approach. However, a key challenge with this

option is ensuring consistency between the transactional and analytics environments. For example, if a payment is not processed successfully, how can they be sure that the transactional layer and analytics layer have a consistent state?

- *Coupled (e.g., database replication):* Database replication technology is utilized between the transactional and analytics components. This option has the benefit of consistency being easier to manage, particularly considering that the TFX system primarily uses SQL databases in most of its core transactional components. It can also be implemented relatively quickly by using standard features that databases provide. However, the schemas of the transactional and analytics environments are more tightly coupled.

One key challenge with the database replication option is that the transactional and analytics components are tightly coupled. The team will not be able to evolve these components independently. For example, if they decide to change the underlying database technology of the Counterparty Manager, changing how database replication works will not be a trivial exercise. If they chose this option, the team could note the tight coupling as technical debt. As the software product evolved, they would keep track of this decision and evaluate whether the approach should be changed.

The tenant and cross-tenant analytics have totally different objectives. In both cases, the underlying technical requirement is the same: either the financial institution using the platform or the owner of the TFX platform wants to generate analytics and benchmarking so that it can provide better advice to its clients. For analytics and benchmarking, anonymization of the data is required. As a result, the team decides to create a separate analytics component to enable these capabilities. If the TFX team had gone down path C for the initial decision, they would have ended up with a structure like that depicted in [Figure 3.7](#).



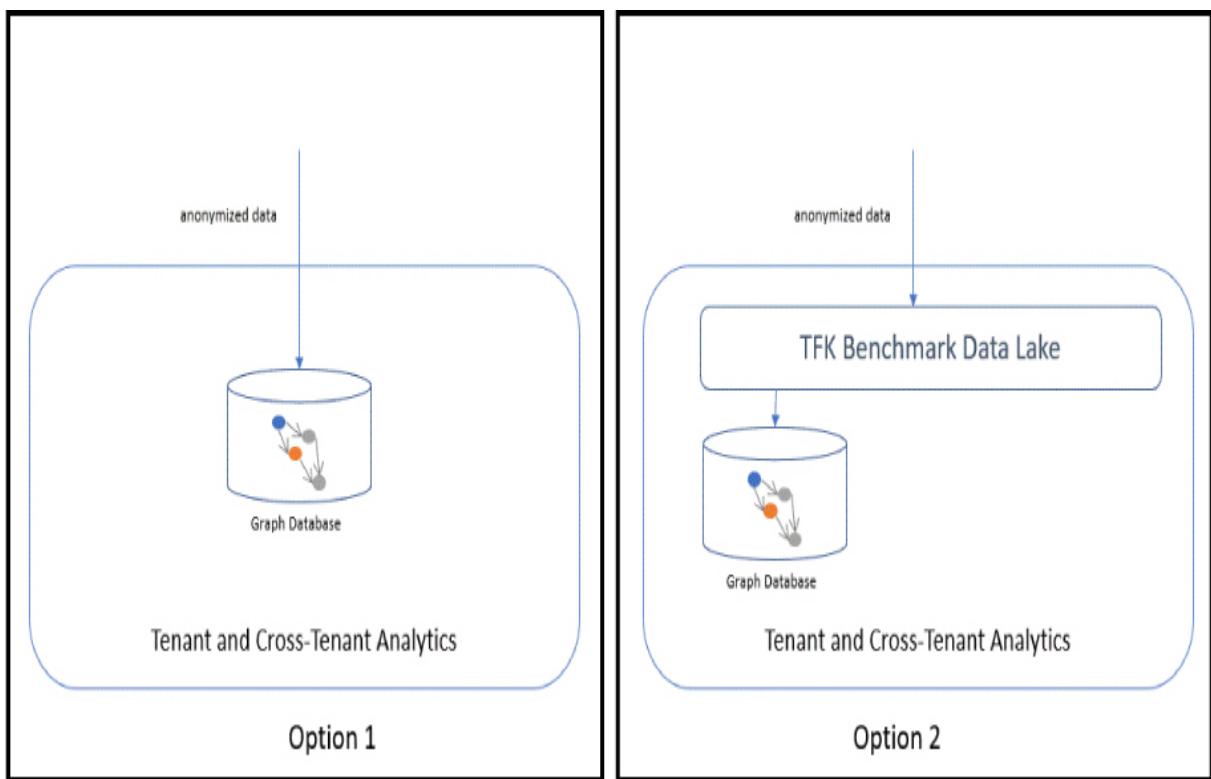
## Analytics

**Figure 3.7 TFX analytics approach**

This approach enables both the client analytics and tenant/cross-tenant analytics to evolve separately and address their own functional and quality attribute requirements. At the beginning of the TFX journey, the tenant/cross-tenant analytical requirements are not that well known. This is a common challenge for most software systems. Let us investigate how the TFX team addresses this challenge from a data architecture perspective based on a particular scenario.

The first set of requirements presented by the product managers involves looking at relationships among exporters, industries, and financial standings. The team might conclude that graph database technology is best suited to deal with such queries. How should they then architect the tenant and cross-tenant analytics components? After several conversations, the team highlights two architectural options.

- Option 1 is to ingest the anonymized data directly into a graph store. This option allows a quick time to market and addresses currently known requirements.
- Option 2 is to create a component that can store each event in an anonymized manner into a TFX benchmark data lake (see [Figure 3.8](#)). A driver to choose this option would be principle 3, *Delay design decisions until they are absolutely necessary*. The team recognizes that different types of questions require different approaches to analyzing the data and potentially different database and analytics technologies. By capturing the data in its basic form, the team effectively buys time in making those choices but are extending their initial delivery time and effort.



**Figure 3.8** TFX cross-platform analytics options

If the team chooses option 2, they can build out additional components for different types of queries. They can also remove the graph database if it is not providing valuable insight. The logic for the team choosing option 1 would be that graph technology is best suited for known use cases and building a benchmark data lake creates unnecessary complexity in the architecture by adding additional components. For example, it requires additional capabilities for managing data integration between different components. This option also increases delivery time and effort. This dilemma is familiar to most experienced software technologists. Architecture is about making difficult tradeoffs and knowing when and why you make them.

Note that using any new database, such as a graph store, adds an additional technology that the team needs to be familiar with. However, since we apply principle 6, *Model the organization after the design of the system*, these decisions are localized within the teams that deal with a certain component, in this case, the tenant and cross-tenant analytics team.

Finally, similar to the data integration decision made between transactional components and the analytics manager, a decision has to be made on how to integrate data between the analytics manager components and the tenant and cross-tenant analytics component. This is a different type of decision given the need to implement anonymization. In addition, the team is aware that the company wants to offer the TFX system to multiple clients. This means that the cross-tenant analytics will require data sourced from multiple tenants of the TFX platform. If we assume a full replication model for multitenancy, it will result in integrating data from multiple instances. We do not explore this topic further, but we emphasize that such a pattern is similar to how data pipelines are managed in analytics environments. We are essentially talking about moving data from different sources to a target with a level of transformation in between. Depending on your environment, this task can get quite complex and is an expanding area of technologies and approaches.

## Additional Architectural Considerations

We conclude this chapter by discussing three architectural considerations related to data: data ownership, data integration, and schema evolution.

### Data Ownership and Metadata

Data flows through our applications. It is shared among different components, where each component interprets the data for its own purpose. For every application, there are a few concepts that are central to the business domain of interest. For example, in the TFX platform, each component needs to understand the buyer- and seller-related counterparty data elements as well as the L/C.

From an architectural perspective, given that key data entities in the system are required by most components, it is important to clearly delineate ownership of the data elements and decide how common data is shared. By ownership, we mean defining systems that are the authority of a particular data element, also called the *master* or single point of truth (SPOT).

This becomes more relevant for larger systems or systems of systems. Neglecting proper data ownership can result in different components

interpreting the data in different ways—in the worst case, resulting in inconsistency in business values.

Let us go back to the components we have in our TFX system and think about them from a data-first perspective. It is important to understand which component is going to **master** which set of data. One simple but effective architectural practice is to create a table of the key data entities and map them to services, as shown in [Table 3.3](#). If you want, you can detail further with a traditional CRUD (create, read, update, delete) view. However, we believe this table is sufficient for managing dependencies.

**Table 3.3 TFX Data Ownership**

Main Data Entities	Document Manager	Contract Manager	Counterparty Manager	Fees and Commissions Manager	Payment Service
L/C terms		Master	Consume	Consume	
L/C document	Master	Consume			
Good		Master	Consume		
Buyer/seller		Consume	Master	Consume	
Bank		Consume	Master		Consume
Payment			Consume		Master
Fees and commissions		Consume		Master	

Looking at a system from this data-first perspective enables us to see which components master a lot of data and which are consumers of data. As a result, we clearly understand data dependencies of the overall system. In the simple example in [Table 3.3](#), it is obvious that the Counterparty Manager and Contract Manager master most of the major data elements; the others are predominantly consumers. The Document Manager and Payment Service are the components that master only a few elements and have limited dependencies on other components. By managing data dependencies, the team ensures loosely coupled components and evolvability of the system, which supports applying principle 4, *Architect for change—leverage the “power of small.”* In our case study, the team has taken the approach of one component mastering a data class.

Once a decision is made that a data entity is owned by a single component, we must decide how the data should be shared with other components. It is important that a data entity is always interpreted consistently across the entire system. The safe way to accomplish this is to share data by reference, that is, by passing an identifier that uniquely identifies the data element. Any component that needs further detail regarding a data entity can always ask the data source for additional attributes that it might need.

Whenever you have a system with distributed data, you also have look out for **race conditions**, where the data can be inconsistent due to multiple updates to the same data set. Passing data by value increases the possibility of this happening. For example, let us assume that a component (e.g., the UI Service) calls the Fees and Commissions Manager to calculate fees and commissions for a certain seller. This requires data about both the seller and the contract. If only the relevant identifiers for the seller and contract are passed, the Fees and Commissions Manager can then call the Contract Manager and Counterparty Manager to get other attributes required for the calculation. This guarantees that the data about the seller and contract is always represented consistently and isolates us from any data model changes in the Contract Manager and Counterparty Manager. If additional data attributes (e.g., contract value, settlement date) were added to the original call, it could create a data-consistency risk because the details of the contract or seller could have been modified by other services, particularly if the UI Service had cached the values for a period of time. We can assume this is a minor risk given the nature of the TFX business process, but it is an important consideration in distributed systems.

As can be seen from this example, referring to data entities by reference creates additional communication between components. This is a significant tradeoff that can add extra read workload on the databases, which can be crippling to a system if not managed effectively. Another way to look at it is to say that we have a tradeoff between modifiability and performance.

Let us assume that the TFX team runs performance tests and discovers that calls from the Fees and Commissions Manager cause extra load on the Counterparty Manager. They can then decide that the attributes for the seller required by the Fees and Commission Manager are limited and are already present in the UI. As discussed previously, the risk of inconsistency (i.e., the attributes being changed between the call to the Fees and

Commission Manager) is low. Consequently, they can decide to pass those attributes directly to the Fees and Commission Manager.

In this example, to meet performance requirements, the TFX team decides not to strictly adhere to passing information only by reference. Instead, they pass all the attributes required (i.e., passed data) by value. However, this is done as an explicit decision.

In this example, we also mentioned that the risk of inconsistency between updates is low. Managing multiple updates to the same dataset from different components in a distributed system can get complex. For example, how should the team manage a scenario where two different components want to update the same contract in the Contract Manager? This can naturally happen in the TFX case study if we assume the importer and exporter are updating attributes of the contract at the same time. If both UIs were accessing the same database, traditional database techniques such as locks could be used to avoid a conflict. However, in the case of TFX, the database is not shared—the UI Service for both importer and exporter can retrieve the same version of the contract from the Contract Manager and independently try to submit updates via an API call. To protect against inconsistent updates, the team would need additional logic both in the Contract Manager and the UI Service. For example, the Contract Manager could check the version of the contract that an update request is made against. If the version in the database is more recent, then the data must be returned to the UI Service with the latest information provided, and a resubmission must be requested. The UI Service will also need additional logic to handle this condition. Distributed systems give us flexibility and resilience, but we also need to make our components deal with unexpected events. We cover resiliency in [Chapter 7, “Resilience as an Architectural Concern.”](#)

Before leaving the topic of data ownership, we briefly touch on the topic of **metadata**, which, as every technologist knows, is data about data. It basically means that you have business-specific data, such as attributes associated with a contract. Then you have data describing either the attributes or the entire contract, such as when it was created, last updated, version, and component (or person) who updated it. Managing metadata has become increasingly important, particularly for big data systems and

artificial intelligence. There are three main reasons for this growing importance.

First, large data analytics systems contain data from multiple sources in varying formats. If you have sufficient metadata, you can discover, integrate, and analyze such data sources in an efficient manner.

Second is the need to track data lineage and provenance. For any output generated from the system, you should be able to clearly identify the journey of the data from creation to the output. For example, in the TFX system, let us assume the team has calculated an attribute in the Analytics Manager that tells the average payment amount for a particular seller against multiple L/Cs. They should be able to clearly trace how that attribute was calculated and all data attributes from each service (e.g., Counterparty Manager, Contract Manager) that was required as an input to the calculation. This sounds easy for simple data flows, but it can get complex in large data analytics environments and systems of systems. Metadata is what makes addressing such a challenge much easier.

Finally, as highlighted by principle 5, *Architect for build, test, deploy, and operate*, Continuous Architecture emphasizes the importance of automating not only software development but also data management processes, such as data pipelines. Metadata is a strong enabler of such capabilities.

## Data Integration

The previous section presented a data-centric view and touched briefly on data sharing. Let us now look at data integration in more detail. The topic of data integration covers a wide area, including batch integration, traditional extract–transform–load (ETL), messaging, streaming, data pipelines, and APIs. Providing a detailed overview of integration styles is beyond the scope of the book. At a high level, there are two reasons for data integration:

- Integrating data between components to facilitate a business process, such as how data is shared between different services in the TFX system. This is where we would find messaging, APIs, remote procedure calls, and sometimes file-based integration. We could also architect our system by utilizing streaming technologies, which would be a different approach to data integration.

- Moving data from multiple sources into a single environment to facilitate additional capabilities such as monitoring and analytics. This is where we traditionally used ETL components, and these days, we consider solutions such as data pipelines.

This section focuses on APIs to provide some examples of how to think about the first type of integration. The reason for this section is to provide a glimpse into the larger world of integration and to emphasize the importance of thinking about integration from a data-centric perspective. We focused on APIs mainly because of their dominance in the industry as an integration style. The same approach can apply easily to messaging. As we explain in [Chapter 5](#), the team can architect the TFX system to switch between synchronous and asynchronous communications based on the tradeoffs they want to make.

Let us start with a brief look into the Semantic Web and resources. The Web is the most successful distributed system that is scalable and resilient. It is built on distributing data via a common protocol, the Hypertext Transfer Protocol (HTTP). The Semantic Web (sometimes referred to as *linked data*) is a method of publishing structured data so that it can be interlinked and become more useful through semantic queries. It builds on standard Web technologies such as HTTP, the Resource Description Framework (RDF), and URIs, but rather than using them to serve Web pages for human readers, it extends them to share information in a way that can be read automatically by computers.<sup>18</sup> Tim Berners Lee<sup>19</sup> defined four principles of the Semantic Web:

<sup>18</sup> [https://en.wikipedia.org/wiki/Linked\\_data](https://en.wikipedia.org/wiki/Linked_data)

<sup>19</sup> Tim Berners-Lee, *Linked Data* (last updated 2009), <https://www.w3.org/DesignIssues/LinkedData.html>

1. Use URIs (Universal Resource Identifiers) as names for things.
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
4. Include links to other URIs so that they can discover more things.

The Semantic Web approach has not become as widely used in the industry as originally anticipated. However, the concepts behind the approach are very powerful and provide insights into how to deal with data integration. At the core of data distribution is the concept of a resource. A resource is a data structure that you can expect to be returned by a Web endpoint.

Representational state transfer (REST) is the most prevalent integration approach that relies on the concept of a resource. The huge success of REST (compared to former integration approaches such as Simple Object Access Protocol [SOAP]) is at least partly because it is an architectural pattern rather than a tightly specified standard or a specific set of technologies. The initial elements of the REST architectural pattern were first introduced by Roy Fielding in 2000,<sup>20</sup> and it primarily consists of a set of architectural constraints that guide its users to use mature and well-understood Web technologies (particularly HTTP) in a specific way to achieve loosely coupled interoperability. Two key benefits REST brought to integration was to change the focus on interfaces from a verb-centric view to a noun-centric view and to eliminate complex, centrally managed middleware such as enterprise service buses (ESBs). In essence, rather than defining a very large set of verbs that could be requested (e.g., `get_customer`, `update_account`), a set of very basic verbs (HTTP methods, e.g., GET, PUT, POST, DELETE) are executed on a noun—which is a resource such as Account. However, it is important to note that there are several different ways in which you can model and build a successful REST API, which goes beyond the scope of this book.

<sup>20</sup> Fielding R. Architectural styles and the design of network-based software architectures. University of California, Irvine, Dissertation.

<http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

Although RESTful APIs are widely used, they are not the only approach you can take. Two examples of other approaches are:

- GraphQL<sup>21</sup> enables clients to ask for a specific set of data in one call rather than in multiple calls, as required in REST.
- gRPC<sup>22</sup> is a lightweight, efficient manner to conduct remote procedure calls via defined contracts.

<sup>21</sup> <https://graphql.org>

<sup>22</sup> <https://grpc.io>

For the TFX system, the team decides that they will expose a set of REST APIs externally to enable partners and clients to access their information. Exposing your data via APIs is an extremely common mechanism used by platform providers not only for technical integration but also to support different business models that support revenue generation, known as the *API economy*. The choice of REST is driven mainly by the wide adoption of REST in the industry.

However, as mentioned earlier, this does not mean that every API has to be RESTful. For example, RESTful APIs can create a chatty interface between mobile components and services that provide the data. This is where an approach such as GraphQL can provide value. However, the team does not see a need to use GraphQL in TFX because it has the UI Service. The objective of the UI Service is to ensure that there are no chatty interactions by creating an API that is tailored to the UI needs.

The topic of integration in general is incredibly wide, and it is not a core focus of this book. It is important that when you think about your data architecture, you focus not only on how you store and manage your data but also on how you integrate it.

## Data (Schema) Evolution

The way we represent and share data evolves along with our software product. How we manage this evolution of our data is commonly referred to as *schema evolution*. Schema evolution can be considered from two perspectives: within a component (intercomponent) and between components (intracomponent).

Within a component, the schema is how the database represents data to the application code. When you implement a database, you always make decisions around entities, fields, and data types, that is, you define a schema (formally or informally). SQL databases have strictly defined schemas, whereas NoSQL databases do not impose as much rigor up front. However, even when you implement a NoSQL database, you still make modeling decisions on how your entities are represented and their relationships—

resulting in a schema even if the database has no manifestation of these decisions.

With time, you realize that you need to enhance your initial understanding of the data, introducing new entities, attributes, and relationships. Within a component, managing how the application code deals with the changes in the underlying data structures is what schema evolution is about. Backward compatibility is a commonly used tactic in this context and basically means that older application code can still read the evolving data schema.

However, there is a tradeoff in increased complexity of application code.

Architecting for build, test, deploy, and operate (principle 5) helps in dealing with this challenge. Wherever possible, we should treat the database schema as code. We want to version and test it, just as we would do with any other software artifact. Extending this concept to datasets used in testing is also extremely beneficial. We discuss this subject in more detail when we cover artificial intelligence and machine learning in [Chapter 8, “Software Architecture and Emerging Technologies.”](#)

Between components, schema evolution focuses on data defined in interfaces between two components. For most applications, this is an interface definition implemented in a technology such as Swagger that implements the OpenAPI standard.<sup>23</sup> For intracomponent schema evolution, concerns similar to those of intercomponent schema evolution apply. As the data being distributed from a component evolves, we want to make sure that consuming components can deal with the changes.

<sup>23</sup> API Development for Everyone (<https://swagger.io>) and The OpenAPI Specification (<https://www.openapis.org>).

Let us now look at a few key tactics that help us deal with schema evolution from an intracomponent perspective. The same tactics apply to other interface mechanisms as well as to intercomponent schema evolution.

First stop is Postel’s law, also known as the robustness principle: *Be conservative in what you do, be liberal in what you accept.*<sup>24</sup> In other words, if you are a producer of an API, you should comply with specific formats and schemas. By contrast, if you are a consumer of an API, you should be able to manage dealing with unknowns in the APIs that you consume from. For example, you should be able to ignore any new fields

that you are not consuming. We cover Postel’s law in more detail in our original *Continuous Architecture* book.

<sup>24</sup> John Postel (Ed.), *RFC 793: Transmission Control Protocol—DARPA Internet Protocol Specification* (Information Sciences Institute, 1981).

By applying Postel’s law, additive data fields become manageable in schema evolution. Consumers that require those fields can consume them, and consumers that do not need them can just ignore them. The challenge comes if the API producer needs to remove certain data fields or significantly alter the schema, commonly known as *breaking changes*. This leads us to the topic of versioning and how we manage change.

As a producer, we need to version our APIs (i.e., our schema). Versioning enables effective management of change by providing a mechanism to communicate with data consumers. Any significant change in the data schema of the API will have to be managed over a period of time. In today’s distributed application landscape, it is not reasonable to expect that all consumers of an API can deal with a breaking change at the same time. Consequently, as a producer of an API, you will need to support multiple versions. Consumers that require the new version for their own functionality can start consuming directly, and other consumers are given time to change to the new version.

This evolutionary way to deal with changes is commonly called the *Expand and Contract* pattern. In this pattern, you first expand to support both old and new versions of the schema. Then you contract and support only the new version of the schema. There are multiple ways of implementing this pattern depending on the technology stack you are using.

In summary, to address schema evolution, you need to manage your data representation as a software asset. This can be a traditional database schema or an API representation. Then you have to create mechanisms for dealing with significant change to support the consumers of your data.

## Summary

This chapter covered how Continuous Architecture approaches data, the primary reason for the existence of our systems. This is an incredibly wide

topic and is an area of increasing technology evolution. We started by briefly delving into the importance of data and highlighting how data architecture decisions can have significant impact on the evolvability of the system and its quality attributes. We touched on the importance of having a common language in a team and referred to how Domain-Driven Design addresses this area.

We then turned our focus to key trends in the data technology space: NoSQL and polyglot persistence, scale and availability: eventual consistency, and event sourcing and analytics. Given the scale of the topic, we summarized key points and highlight areas of applicability for the TFX case study.

In particular, we discussed how polyglot persistence can be applied to TFX, where each key component has its own database (principle 4, *Architect for change—leverage the “power of small”*) and can decide on the appropriate technology to meet its quality attributes (principle 2, *Focus on quality attributes, not on functional requirements*). We also discussed how evolving requirements can result in an additional data component and store for goods delivery and the choice of a document store solution to deal with the unknowns in that space. On the topic of analytics, we looked into the required architecture decisions and discussed the applicability of principle 3, *Delay design decisions until they are absolutely necessary*, and principle 6, *Model the organization of your teams after the design of the system you are working on*.

We wrapped up the chapter by discussing wider architectural considerations for data from the perspective of Continuous Architecture: data ownership, data integration, and data (schema) evolution. In summary, you need to know where your data is and how it is managed (data ownership), how you are going to share that data (data integration), and how to manage the data over time (data evolution).

The key points from this chapter can be summarized as follows:

- Focus on developing a common language between the team and the business stakeholders. Domain-Driven Design is a good approach that promotes a common language and enables development of loosely coupled services.

- The common language should live within your code and other artifacts, including not only the data aspects but also.
- There is a wide variety of database technologies to choose from. Evaluate them carefully, because changing will be an expensive task. Your selection should be based primarily on the quality attribute requirements, particularly the tradeoff between consistency and availability.
- Data is not an island. All data needs to integrate. Be clear which components master the data, how you refer to the data (i.e., identifiers), and how you evolve its definition over time.

The next few chapters focus on different quality attributes, starting with security, and their impact on the architecture as well as their associated architectural tactics.

## Further Reading

This section includes a list of books that we found helpful to further our knowledge in the topics that we have touched on in this chapter.

- Although Domain-Driven Design is not solely about data, we referred to it as an approach that supports the concept of a ubiquitous language and introduced the concept of a bounded context. The seminal book in this topic is *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley, 2004) by Eric Evans. Vernon Vaughn has also written extensively on this subject. He offers helpful insights in *Implementing Domain-Driven Design* (Addison-Wesley, 2013) and provides a good introductory book in *Domain-Driven Design Distilled* (Addison-Wesley, 2016).
- A comprehensive book on a broad range of database and distributed data processing technologies is *Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems* (O'Reilly Media, Inc., 2017) by Martin Kleppmann. It has a strong focus on quality attributes and explains in detail how technologies address distributed data challenges.

- If you are more interested in tactics that enable you evolve your database design, *Refactoring Databases: Evolutionary Database Design* (Pearson Education, 2006) by Scott Ambler and Pramod Sadalage is a good reference book.
- There are several books on NoSQL databases, most going into specifics of certain technologies. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence* (Pearson Education, 2013) by Pramod Sadalage and Martin Fowler provides a good basic overview. Dan Sullivan's *NoSQL for Mere Mortals* (Addison-Wesley, 2015) covers the same topic but in more detail.
- If you want to find out more about event processing, you can look into *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems* (Addison-Wesley, 2002) by David Luckham and *Event Processing in Action* by Opher Etzion and Peter Niblett (Manning, 2011).
- Although data science is not the focus of this book, if you want to gain knowledge about it, we recommend *Data Science for Business: What You Need to Know about Data Mining and Data-Analytic Thinking* (O'Reilly Media, 2013) by Foster Provost and Tom Fawcett and *Data Science* by John Kelleher and Brendan Tierney (MIT Press, 2018).
- We did touch on the importance of APIs and highlighted the importance of designing good APIs. There are several detailed technical books on this topic. *The Design of Web APIs* (Manning, 2019) by Arnaud Lauret provides a higher-level overview on key concepts from the perspective of Web APIs. If you are interested in integration in general, a classic book on the topic is *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* by Gregor Hohpe and Bobby Woolf (Addison-Wesley, 2004).

# Chapter 4. Security as an Architectural Concern

*If you’re going to defend systems against attack, you first need to know who your enemies are.*

—Ross Anderson

Only a few years ago, security was viewed by many software architects as an arcane subject that needed to be seriously addressed only for specialist systems such as those handling payments or for use by the intelligence community. How things have changed in a short time!

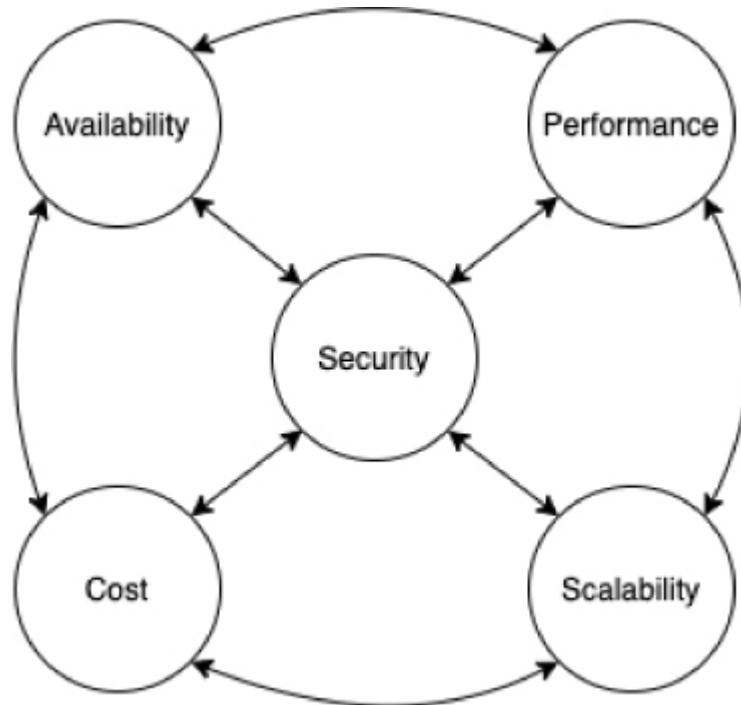
Today, security is everyone’s problem. It is a critical concern for every system given the threats that modern Internet-connected systems face and the stringent security-related regulations, such as the General Data Protection Regulation (GDPR) privacy regulations, that we all need to comply with. In this chapter, we examine how to approach security as an architectural concern, recap the common security mechanisms we need to be familiar with, and consider how to keep our systems secure when in operation.

Our case study, the Trade Finance eXchange (TFX) system automates a sensitive business process, is Internet-connected, and is intended to be a multitenant system at some point, so it is likely to have quite demanding security requirements. Much of the data contained in the TFX system is commercially sensitive, making it a valuable target for malicious adversaries to try to obtain. Consequently, protection of the data is important. Disrupting the system’s operation could provide commercial advantage to some organizations or a promising avenue for an extortion attempt against TFX’s owners, users, and operators. Customer organizations are likely to be asking a lot of questions about the security of TFX, so the team must seriously consider security from the beginning of the development of the system.

# Security in an Architectural Context

Computer security is a broad field, with many specialist roles dedicated to it. Given its complexity and criticality, it is important to involve security experts in your security work. Our role is to achieve a balance between security and the other architectural concerns in our specific situation and to be sufficiently knowledgeable about the topic to ensure that our architectures are fundamentally secure and to know when to involve the experts to provide validation and their specialist knowledge.

Of course, security does not exist in isolation but is one of a number of important quality attributes for most systems. All of the quality attributes are interrelated and require tradeoffs among them to find the right balance for a particular situation. [Figure 4.1](#) illustrates this interrelatedness in a simple diagram.



**Figure 4.1** *Security in context*

What [Figure 4.1](#) shows is that making a system secure can have an impact on almost any other quality property, a few of which are availability, performance, scalability, and cost, which themselves are interrelated too, as we'll show in the later chapters of the book.

Making a system secure can add processing overhead to many functions of the system, thereby reducing the performance—as well as scalability—of the system. Specific pieces of the security technology itself also may have scalability limitations.

Generally, security and availability have a positive relationship, as improving security helps to make the system more resilient and so improves its availability. However, ensuring security in failure cases can also make the system less available because of factors such as the need to establish trust relationships between components on restart, which can slow things down or cause the system to not start up if the process fails.

Most quality attributes increase cost, and security is no different, as it adds complexity to both building and operating a system and may involve purchasing security technologies or services that increase the direct costs of the system.

None of this is to suggest that we should try to avoid security or cut costs by limiting its effectiveness. Rather, it is just a reminder that nearly every architectural decision is a tradeoff, and it is important to be aware what tradeoffs we are making.

## **What's Changed: Today's Threat Landscape**

Only a few years ago, many large software systems lived relatively isolated lives, running in tightly controlled data-center environments with access limited to employees of the organization that developed it. These systems faced quite limited threats to their security owing to the limited access available to them. Today, it goes without saying that the threat landscape that our systems face has changed dramatically.

Most of today's software systems are Internet-connected, and many run in a highly distributed manner on public cloud platforms, integrating services from across the Internet. This means that the security environment today is that of Internet security rather than the simpler and safer in-house environment.

As a cursory glance at a security news website or any of the live threat maps<sup>1</sup> reveals, the Internet is a hostile security environment for a software

system, and the level of security threats that systems face has risen sharply in recent years, with no sign of abating.

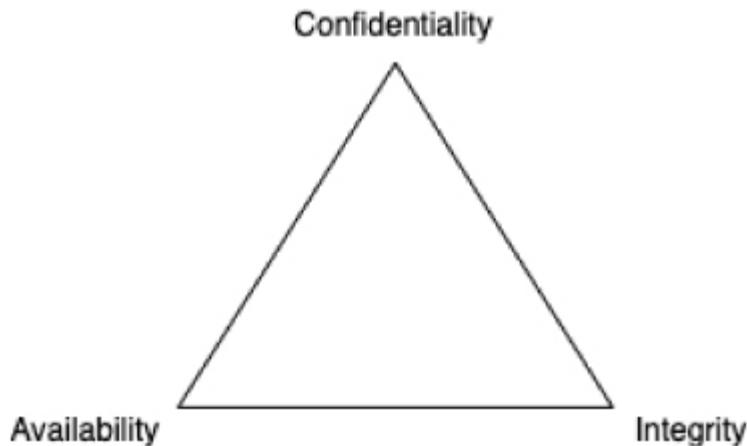
<sup>1</sup> Such as those provided by Kaspersky, FireEye, and Norse, to name a few examples.

Network-connected systems, the large number of professional (and amateur) malicious actors looking for vulnerabilities, the use of public cloud computing, the large amount of externally developed commercial and open source software in our **software supply chains**, limited general security awareness by many software engineers, and the dark net industry packaging security exploits and attack mechanisms into easy to use services (such as **botnets** for rent) all combine to form a dynamic and hostile environment for a system to operate within.

## What Do We Mean by Security?

Security is a broad field encompassing many subfields, affecting the entire system delivery life cycle, including security requirements, secure infrastructure design, secure operation, secure software development, and secure system design. Security specialists must understand myriad principles, jargon, security mechanisms, and technologies. For our purposes, it is important that we keep focused on the core objectives of security and make sure we understand what to prioritize for our systems.

As you will undoubtedly remember, the objectives of security are confidentiality, integrity, and availability, or the *CIA triad*, as they are commonly referred to and represented, as shown in [Figure 4.2](#).



**Figure 4.2** The confidentiality, integrity, availability (CIA) triad

Confidentiality means limiting access to information to those who are authorized to access it; integrity means ensuring that only valid changes can be made to the system's data and that the changes can be initiated only by suitably authorized actors; and availability means ensuring that the system's data and functions are available whenever they are needed even in the face of malice, mischance, or mistakes.

From an architectural perspective we must ensure that we understand the requirements for confidentiality, integrity, and availability; the threats that the system faces that could compromise those objectives; and the set of security mechanisms (people, process, and technology) that can be used to mitigate those threats and achieve the requirements.

Capturing security requirements can be quite difficult. Some security requirements are actually specialized functional requirements, and these can be captured as user stories, perhaps augmented by techniques such as user access matrices, to define the access required by each user role, and the quality attribute scenarios that we discussed in [Chapter 2](#). However, in many cases, we are trying to indicate what should not happen rather than what should happen, and, unlike many quality attributes, most security requirements are difficult to quantify. An alternative is to use a checklist-based approach to provide a set of checks to perform on your system to avoid security problems. An example of such a checklist is the Open Web Application Security Project (OWASP) Application Security Verification Standard (ASVS).<sup>2</sup>

<sup>2</sup> Open Web Application Security Project (OWASP), *Application Security Verification Standard*. <https://owasp.org/www-project-application-security-verification-standard>

In summary, for any real system, ensuring it is secure is a complex process involving tradeoffs against other important quality attributes (such as performance and usability).

## Moving Security from No to Yes

The security specialists in many large organizations are gathered into a central security organization whose role appears to be as the group who always says no, or the group that rarely says yes! At least, that is how it

may seem to those outside the group, trying to deliver change in the IT environment.

Of course, this is a rather unfair characterization of a group of skilled, professional security engineers who are often consulted far too late and are often underresourced for the size of problem they are responsible for. They are frequently forgotten when everything goes well but immediately blamed for any security problem anywhere in the organization. No wonder they tend to be conservative and to say no more often than yes when people are asking to make major changes to systems.

Times are changing, and many security groups are becoming more proactive, more aligned to the teams developing systems, more focused on balancing risk rather than preventing mistakes at any cost, and valuing speed of change as well as risk mitigation. The part we must play in this evolution is to make sure that we integrate security throughout the delivery process (see next point) and treat security specialists as our partners in achieving a crucial quality attribute rather than as adversaries to be negotiated around.

## Shifting Security Left

Part of achieving a cooperative relationship with our security engineering colleagues is to integrate security considerations into the entire system delivery life cycle, viewing security as work that is done in small, frequent steps (“little and often”), not relegated to the end of the process where it becomes an obstacle to be surmounted rather than a crucial attribute of our system.

Integrating security work into the entire life cycle is often referred to as *shifting security left*, meaning that we move security as early in the process as possible, which is one of the principles of a **DevSecOps** approach.

Shifting security left involves early consideration of security requirements, a consideration of security threats and their implications as system architecture work emerges, a constant focus on security as we code and test, and a continual process of security review and testing as the system is developed. In this way, security is engineered along with the other development tasks.

Of course, some aspects of security work, such as penetration testing, where specialist testers attempt to break the system's security controls from outside, may make more sense late rather than early in the life cycle, but the principle is that we still address security as early as possible. Other tasks, such as threat modeling and security design reviews, can be started as soon as we have architecture and design proposals to analyze, and static code analysis should be started as soon as code is created.

## Architecting for Security

When architecting a system, we must understand its security needs, the security threats that it faces, and the mainstream mitigations for those threats to ensure that we create a system with sound security fundamentals. This section discusses how to identify the threats, the tactics and mechanisms for mitigation, and how to maintain security in live systems.

### What Is a Security Threat?

For many years, the security field has wrapped itself in its own terminology, building a certain mystique along the way, and the language of “threats” and “threat modeling” is an example this. The idea of threats and mitigating them is actually very natural to us as human beings; we just don’t talk about them in this way.

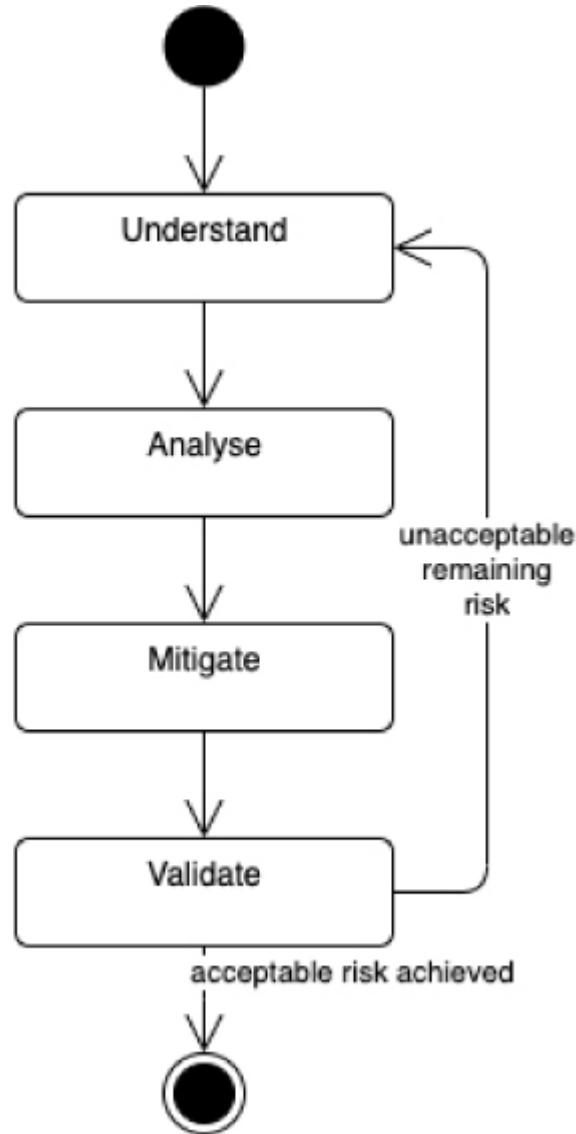
The threats that our system faces are simply those things that could go wrong and cause us a security problem, often, but not exclusively, as a result of a malicious actor who wishes to attack the system in some way. Similarly, threat modeling is the activity of identifying those things that can go wrong (threats) and working out how to mitigate them.

### Continuous Threat Modeling and Mitigation

Today, most software engineers and architects are not familiar with the concept of threat modeling and typically don’t spend time thinking systematically about the security threats that their systems face. Threat modeling sounds like a mysterious and complex technique best left to security engineering specialists, whereas it is actually a simple idea that is relatively easy to perform (at least when compared to many of the complex

activities involved in modern system delivery). Principle 2, *Focus on quality attributes, not on functional requirements*, helps us to remember to start threat modeling early to ensure sufficient focus on security.

The threat modeling and mitigation process involves four phases: understand, analyze, mitigate, and validate, as shown in [Figure 4.3](#). These steps combine to help us address our security risks:



**Figure 4.3** Steps in threat modeling and mitigation

- **Understand:** Until we thoroughly understand what we are building, we cannot secure it. This usually starts with some architectural sketches to help us understand the scope and structure of what we're building,

including system boundary, system structure, databases, and deployment platform. For the TFX system, the team consults the architectural diagrams in the Information, Functional, and Deployment views (which you can find in [Appendix A](#)).

- *Analyze*: Work through the design of our system and analyze what can go wrong and what security problems this will cause. There are several techniques to help with this process, and we outline two of the most popular models—STRIDE and attack trees—and show how they can be applied to TFX.
- *Mitigate*: Once the threats are identified and characterized, we can start thinking about how to mitigate them. This is the stage at which we start talking seriously about security technologies—**single sign-on (SSO)**, **role-based access control (RBAC)**, and **attribute-based access control (ABAC)**, encryption, digital signatures, and so on—and security processes such as multilevel authorization of sensitive actions and security monitoring and response processes. Software engineers and architects naturally focus more on the technical mechanisms, but the mechanisms must work with the people and the processes, so it’s important to have an awareness of all three aspects of security.

Part of this step is also considering the tradeoffs we need to make when implementing security mechanisms. How do they affect other quality attributes such as performance and usability?

The mitigation step is where many software development teams start the process, yet unless they first understand the threats, it’s difficult to choose the right set of mitigations.

For TFX, the team works through their threat model and starts thinking about how to identify the users, ensure that sensitive operations are checked and protected, limit data access to authorized users, protect data in the databases and at interfaces, respond to a security incident, and so on.

- *Validate*: Having an understanding of the system, the potential security threats, and mitigation strategies, we have made a good step forward, as we now have a threat model and a set of proposed mitigations for

the threats we found. As a result, we have a much better estimation of the security threats our system faces than we had at the beginning.

Now is the time to step back and review everything we've done so far. This is also a good point to ask independent experts, such as our corporate information security team, for their review and input. Do we need to run any practical tests to ensure our decisions are sound? Are the tradeoffs we have made with other quality attributes the right ones? Do we think our threat lists are complete? Are our mitigations effective and practical? Do we know how we are going to test that each threat is mitigated as we are building? Are we happy with any residual risk we think we have left? If we're unhappy with the answers to any of these questions, then we repeat the process (perhaps a few times) until we have satisfactory answers.

Of course, it is hard to know when enough is enough because there is no way of being sure that all the credible threats have been identified. Our suggestion is to consult a few independent experts, make sure that you are reasonably informed about common threats and security practices (use mainstream security websites and books of the sort that we suggest under "[Further Reading](#)" at the end of the chapter), and then use your judgment. After a little practice, you will be able to apply the threat modeling and mitigation process to any system you work on to understand the major threats that it faces and the strategies you can use to mitigate those threats.

Now that we have the first version of the security design for our system, we're in a good position to understand our security risks and to be confident that we have chosen the right the set of mitigations (mechanisms).

However, this is only the first version of the threat model, and as we do for most architectural activities, we want to take the "little and often" approach so that the threat modeling starts early in the life cycle when we have very little to understand, analyze, and secure. Then, as we add features to our system, we keep incrementally threat modeling to keep our understanding up to date and to ensure we mitigate new threats as they emerge; this is an example of principle 3, *Delay design decisions until they are absolutely necessary*. This approach allows threat modeling, and security thinking in general, to become a standard part of how we build our system, and by

going through the process regularly, our team members rapidly become confident and effective threat modelers.

In the next section, we explore some techniques to help us identify the security threats that our system may face.

## Techniques for Threat Identification

A very common problem when people first come to threat modeling is where on earth to find the threats. Some, like **SQL injection attacks**, are perhaps obvious, but until someone is quite experienced, he or she will benefit from some guidance and a structured approach. There are quite a few approaches to threat identification (threat modeling), and we discuss a couple of them in this section.

Helping their engineers identify threats in a structured way was a problem that Microsoft faced some years ago, and the company found it helpful to create a simple model to help people structure their threat identification process (the “what can go wrong” step). The model it adopted and popularized, known as STRIDE,<sup>3</sup> is an example of a threat modeling framework.

<sup>3</sup> Adam Shostack, *Threat Modeling: Designing for Security* (Wiley, 2014).

STRIDE is one of the oldest threat modeling frameworks, and its name is mnemonic constructed from the first letter of each of its components: spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege. Each of these components is a type of threat that endangers one of the CIA properties of our system:

- *Spoofing*: To impersonate a security identity other than your own, thereby violating *authorization* controls and, in turn, confidentiality.
- *Tampering*: To change data in an unauthorized way, violating *integrity*.
- *Repudiation*: Bypassing the identity controls of the system to prevent the attacker’s identity being linked to an action, thereby violating the *nonrepudiation* principle, which is part of integrity.
- *Information disclosure*: Accessing information without authorization to access it, thereby violating *confidentiality*.

- *Denial of service*: Preventing the system or some part of the system from being used, thereby violating *availability*.
- *Elevation of privilege*: Gaining security privileges that the attacker is not meant to have, thereby potentially violating *confidentiality*, *integrity*, and *availability*, depending on the actions taken by the attacker.

Let's consider how we might apply STRIDE to our example system. Even for this relatively straightforward system, a real threat model would comprise a significant amount of information, probably held in a fairly large spreadsheet or similar sort of document. We don't have space here to present more than a fragment of it, but some examples of STRIDE threats that we might identify for the TFX Payment Service are shown in [Table 4.1](#).

**Table 4.1** Example STRIDE Threats for TFX

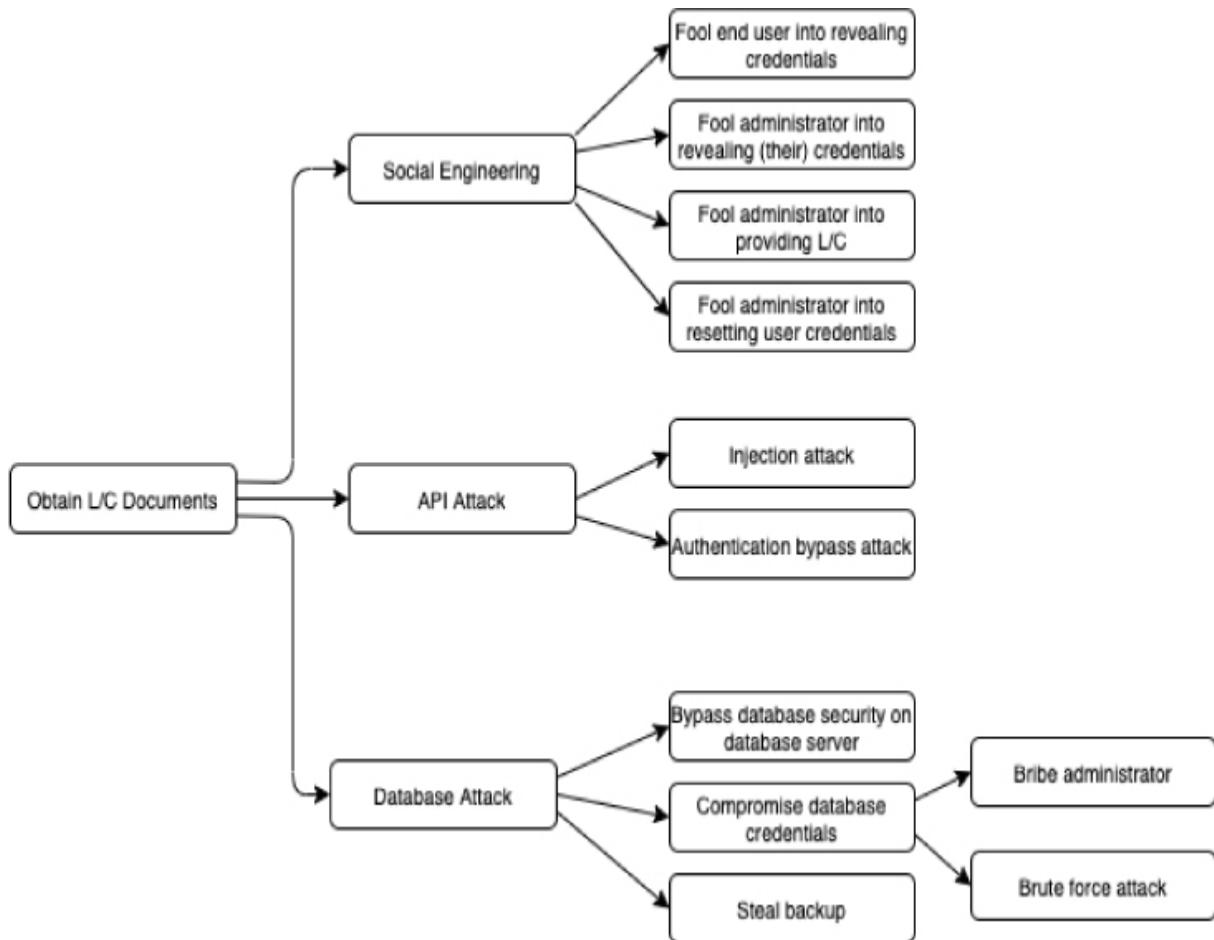
<b>Threat Type</b>	<b>System Component</b>	<b>Description</b>	<b>Tracker</b>
Spoofing	Payment Service	Adversary with access to the network sends unauthorized payment request to attempt to transfer funds illicitly.	TFX-123
Tampering	Payment Service	Adversary accesses the database and tampers with a payment request to have it resubmitted with fraudulent account details.	TFX-127
Repudiation	Payment Service	Customer fraudulently claims that a payment from his or her account was made without the customer's authorization.	TFX-139
Disclosure	Payment Service	Access to the service database allows confidential details of payments made and requested to be accessed.	TFX-141
Denial of service	Payment Service	Adversary manages to drastically slow network connection from Payment Gateway, causing massive buildup of requests from Payment Service, leading to failure.	TFX-142
Elevation of privilege	Payment Service	Administrator from client attempts to send unauthorized payment by guessing supervisor password.	TFX-144

These simple examples illustrate working through representative STRIDE threats from the perspective of a single system component. Each threat is classified by type and system component affected, described briefly, and then tracked in some form of issue-tracking system to allow it to be fully described and its mitigation and validation tracked and reported on.

Of course, you can work both across and down, thinking about all the spoofing threats right across the system, as well as thinking about the different threats that an individual system component is subject to. You are also likely to spot a number of different threats of each type for most system components (e.g., in our example, a second tampering attack is to intercept a valid payment request and change critical information, such as the payment amount or the bank account details). You can see that systematically working through the threats your system might face rapidly causes the kind of healthy paranoia a security engineer needs, and you are likely to end up with a large number of threats to mitigate. This is well and good and reveals effective analysis rather than a problem with the process.

STRIDE is a bottom-up approach to finding threats, by which we mean that it understands how the system works and looks for security weaknesses in its structure and implementation. However, approaches such as STRIDE do not consider potential attackers and their motivations, which vary widely among systems. Thinking about the motivation of a potential attacker can be a rich source of threat ideas, and other threat modeling frameworks, such as attack trees, consider the threats to the system from this perspective.

Attack trees are a well-known and fairly simple technique for threat identification from the perspective of an attacker's motivation. The root of an attack tree is the goal attributed to the attacker, and it is decomposed into child nodes, each representing a strategy for achieving the goal. The child nodes are, in turn, decomposed into a more refined strategy, with the process continuing until the leaf nodes of the tree represent specific, credible attacks on the system. An example of an attack tree for the TFX system is shown in [Figure 4.4](#).



**Figure 4.4** Example attack tree for TFX

This attack tree explores how an attacker might go about obtaining a copy of a letter of credit (L/C), which could have commercial value to the attacker. It identifies three strategies that might be used to obtain the L/C: use social engineering against end users and administrators, attack the system API, and attack the system's database. For each strategy, the tree identifies a set of possible ways that the attack could be performed. This allows us to think through threats from the perspective of a goal being met rather than simply vulnerabilities in our system's architecture and implementation. Some of these threats are unlikely to be credible (e.g., in the case of TFX, directly attacking the database server is very unlikely, as we are using a cloud service for our database), whereas others are credible, but we may need to decompose them further to fully understand them. Once we have worked through the process, we can extract a set of possible threats from the remaining leaf nodes of our tree.

As well as identifying the threats, attack trees can be annotated with risk factors, such as likelihood, impact, difficulty, cost of attack, likely effort, and so on, to allow the threats to be compared and analyzed. In our experience, however, most people use attack trees only to identify the threats.

## Prioritizing Threats

Models such as STRIDE and attack trees help to structure the process of finding potential threats to our system, but for most systems, not all threats are equally serious. Consequently, we need a way of ranking threats to help us consider the importance of mitigating them.

A number of approaches, at different levels of sophistication, have been created for rating the severity of threats, but for our purposes, a very standard risk quantification approach will suffice. We suggest simply considering how likely a threat is to be realized and how serious an impact its realization would be, an approach used in many risk evaluation scenarios, not just security. These are judgment-based estimates, and so a simple Likert<sup>4</sup>-type rating of low, medium, and high for each is sufficient to characterize the threats. For each threat that your threat analysis revealed, you can estimate its likelihood and impact.

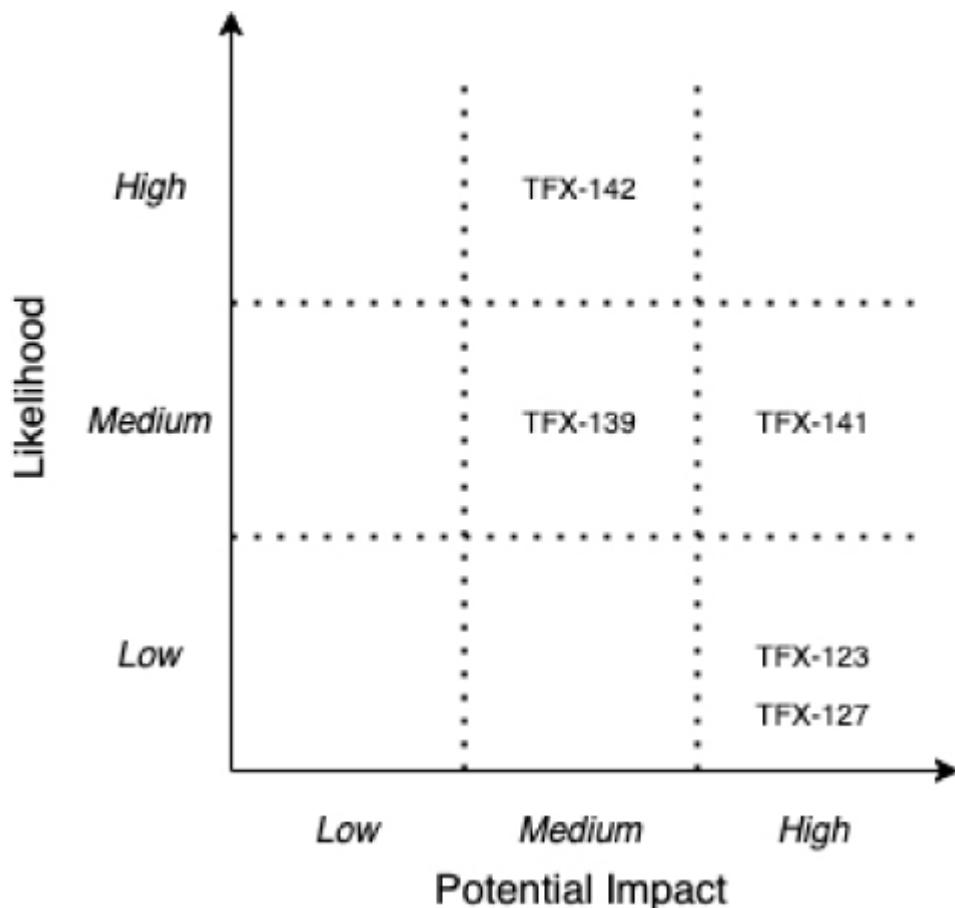
<sup>4</sup> Qualitative rating scales used in questionnaires, originally devised by social scientist Rensis Likert in 1932. <https://www.britannica.com/topic/Likert-Scale>

A high likelihood would be assigned to threats that could be automatically discovered or are well known and that do not require a high degree of sophistication to exploit. A medium likelihood would be assigned to threats that are not obvious and that require some skill or specialist knowledge to exploit. A low likelihood would be assigned to threats that are obscure and difficult to identify or that require a high degree of sophistication to exploit.

A high-impact threat is one that is likely to be highly injurious to the owning organization if exploited (e.g., a major GDPR-related data breach). A medium-impact threat could be one where the sensitivity or scope of the breach is limited, but it still requires major remediation effort and has reputational or organizational impact for the owning organization and affected individuals or organizations. A low-impact threat might be one

where, although there is some degree of loss, the costs to the owning organization and affected individuals or organizations are low

However, it is important to remember that this is a qualitative assessment exercise and is most certainly not science but simply a way of comparing a set of opinions. An example of classifying security threats in this way for our TFX system is shown in [Figure 4.5](#), using the threats outlined in [Table 4.1](#).



**Figure 4.5** Classifying security threats

For example, we might look at TFX-141, an information disclosure risk, and estimate that its likelihood is medium or low but that its impact is definitely high, giving a score of H, M or H, L. If we are a little pessimistic (or realistic, as a security engineer might point out), this would be H, M and would indicate a fairly serious risk, which is where we have placed it in the figure. However, as we said earlier, always remember that experience and

intuition are at least as valuable as a structured but judgment-based approach like this one. If you think a threat is serious, it almost certainly is.

## Other Approaches

Finally, it is important to be aware that there are many other more sophisticated approaches to threat modeling, including Process for Attack Simulation and Threat Analysis (PASTA), Visual, Agile and Simple Threat (VAST) modeling method, Operationally Critical Threat, Asset, and Vulnerability Evaluation (OCTAVE), MITRE's Common Attack Pattern Enumeration and Classification (CAPEC), and the National Institute of Standards and Technology (NIST) model to name but a few. They all have their strengths, weaknesses, and unique aspects. Our general advice is to keep your approach simple and focused on the threats rather than on the details of the threat modeling process. However, as you gain experience in threat modeling, it is useful to understand the differences between the approaches so you can adapt your approach to various situations. We provide references for further reading on threat modeling approaches at the end of the chapter.

The important point is that threat modeling is an extremely useful and relatively straightforward technique, whatever approach you use. Just doing threat modeling puts you in a better position than most of the projects in the industry today and provides a good foundation on which to base your security design and your work with security specialists with whom you may collaborate.

## Architectural Tactics for Mitigation

When we understand the possible threats that our system faces and have some idea of the severity of them, we can identify architectural tactics to apply to our system to mitigate these threats. It is interesting to observe that most teams start considering security at this point, identifying technologies and sometimes design patterns without a clear understanding of what threats they are mitigating. This is often why significant security problems are overlooked.

You are likely already familiar with a range of security tactics and have probably used them many times. Therefore, we aren't going to explain the basics of fundamental security mechanisms such as authorization or encryption. Instead, we briefly define each tactic to avoid any ambiguity and provide a reference for further reading. Then we examine how each of the tactics would be applied in the context of TFX, our example system.

We remind you that security involves people and processes as well as technology, something we talk more about in a later section. As architects, we are usually accountable for the technology decisions rather than for the people and process decisions, but all three are equally important, and we must keep them in mind when considering architectural tactics.

## **Authentication, Authorization, and Auditing**

Three fundamental technical mitigations for security threats are authentication, authorization, and auditing.

We use authentication mechanisms such as usernames and passwords, along with more sophisticated mechanisms such as **two-factor authentication (2FA)** tokens, smart cards, and biometric readers, to confirm the identity of a security principal, that is, a person or system that is interacting with our system. Authentication is a prerequisite to implementing many types of security control and is a specific mitigation for spoofing threats in which the attacker impersonates a trusted source.

We use authorization mechanisms, such as RBAC, to control what actions security principals can execute and what information they can access. These mechanisms mitigate elevation of privilege risks in which the attacker attempts unauthorized access.

We use audit mechanisms, such as securely recording security events such as access attempts, to monitor the use of sensitive information and actions to prove that our security mechanisms are working as intended.

In the TFX system, we obviously need to use strong authentication to identify all the users of our system, probably using 2FA to mitigate against brute force attacks on weak or reused passwords. We also need an authorization mechanism to ensure that we have the ability to control which users can perform which operations on which objects (e.g., releasing a

payment or issuing an L/C). In addition, we need to record all significant business events in an audit trail so that we can investigate unusual behavior or customer disputes.

## Information Privacy and Integrity

Information privacy and integrity are often what laypeople think of first in regard to computer security. This is probably because, for people outside the field, security is primarily associated with cryptography, and privacy and integrity are often achieved using mechanisms such as encryption and hashes and signatures.

For our purposes, we define *information privacy* as the ability to reliably limit who can access data within a system or data passing between the system and other places (typically other systems). By limiting access, we mitigate information disclosure risks.

When we consider protecting information, we usually separate our consideration of information at rest—data held in files and databases—and information in transit—data in network requests, messages, and other forms of transmission. Privacy is usually achieved for information at rest using access control systems (authorization) and secret key encryption (for performance reasons). Information in transit is usually protected using encryption, typically a combination of secret key and public key **cryptography** (**secret key cryptography** is used for performance reasons; **public key cryptography** is used to exchange the secret key encryption key).

From a security perspective, *information integrity* is the mitigation of tampering risks by preventing unauthorized security principals from changing information or authorized principals from changing it in an uncontrolled way. Again, tampering risks exist for information within the system and traveling between the system and other systems. Within the system, an authentication mechanism can help to prevent tampering, but for high-risk situations, it's necessary to use a cryptographic solution such as **cryptographic hashing**, which allows changed information to be spotted because it no longer matches its hash value; or signing, which is a similar operation but using public/private key cryptography, meaning that the

identity of the principal guaranteeing the information integrity can be checked using the relevant public key.

---

### Note

Defects in your application software can compromise privacy and integrity even if cryptographic controls are in place, highlighting the importance of basic software quality as part of achieving security.

In our TFX system, the team must balance complexity and risk to decide on the right level of information privacy and integrity controls to implement. We can refer to our threat model to help make these decisions, but as we noted previously, TFX certainly needs authentication to act as the basis of an information privacy mechanism. Given the commercial sensitivity of much of the information in the system, the team might well decide to encrypt some of the databases in the system (perhaps the document and contract databases), accepting the operational complexity that this adds, such as key management in the database and the need to handle encrypted backups. It is also likely that they would secure connections between our services and databases and certainly those between our system and the outside world, particularly for the payment flows. Further consideration of threats might also suggest the need for cryptographic hashing of documents exchanged with third parties to ensure that tampering can be clearly identified. Each of these steps requires a tradeoff between complexity, cost, and the likely risk posed to the TFX system. In addition, in accordance with principle 3, **Delay design decisions until they are absolutely necessary**, the team would probably delay adding these security mechanisms until they were needed and they could justify the cost and complexity tradeoffs of implementing them.

### Nonrepudiation

A security tactic less familiar to most nonspecialists is ensuring the characteristic of nonrepudiation, which is using mechanisms to prevent security principals from denying their actions. As its name suggests, this tactic is a mitigation for repudiation threats.

Nonrepudiation can also involve cryptography, where requests or data items are signed with the private key of a security principal such as a user or a software system. We can use the public key of the principal later to prove that the principal's private key was used to sign a piece of data, so proving that it originated from that principal. The other common technical mechanism to implement nonrepudiation is logging, where we simply keep a secure, tamper-resistant list of the actions that our security principals have executed so that it can be used as evidence later.

There are also nontechnical approaches to nonrepudiation, such as ensuring the effectiveness and usability of authentication systems (perhaps using mechanisms such as biometrics to avoid the need to remember passwords), so that users cannot share logins or try to bypass authentication checks. Some technical approaches, such as the use of 2FA tokens, can also make it difficult for users to share logins and so strengthen nonrepudiation claims.

Remembering principle 5, **Architect for build, test, deploy, and operate**, we need to consider how the system will be used in its operational environment. Given the nature of the commercial interactions that TFX is automating, nonrepudiation is important. A number of the actors in a trade-finance transaction chain have an obvious interest in being able to plausibly deny having received or agreed the content of a document at a later point, to avoid paying a charge or to try to change their minds about a transaction without incurring any costs, such as a cancellation charge. Realistically, the team would probably start with strong authentication and an audit trail of user actions. But they might well need to provide cryptographic assurance at a later point, signing documents and actions with the private key of the executing user, to provide a higher degree of assurance.

## System Availability

It is sometimes a surprise to newcomers in the security field that availability is an important aspect of security when the field is often characterized as being concerned with topics such as authorization and cryptography. Although closely related to the quality of resilience (see [Chapter 7](#), “[Resilience as an Architectural Concern](#)”), many of the security threats that our Internet-connected systems face today are **denial-of-service** threats, and we mitigate them using availability tactics. Another class of availability

threat that has emerged recently is that of **ransomware**; detection and backup are tactics for mitigating these dangerous attacks.

Denial-of-service attacks overload the system in some way in order to exhaust a limited resource, such as processor power, memory space, storage space, network bandwidth, network connections, or any other limited resource in the system. For Internet-connected systems, the most common type of denial of service is network-based, but application-layer attacks (which target weaknesses in application software, open source libraries, or middleware) are becoming more common.<sup>5</sup> A related attack in the cloud environment is a commercial usage attack, where flooding the system with requests causes it to incur large cloud computing costs, perhaps affecting the commercial viability of the service or the organization.

<sup>5</sup> Junade Ali, “The New DDoS Landscape” [blog post], The Cloudflare Blog, 2017, <https://blog.cloudflare.com/the-new-ddos-landscape>

Architectural tactics for system availability is a broad topic, which we discuss in detail in [Chapter 7](#). However, some of the most common tactics are throttling of incoming requests, based on the load within the system; using quotas to prevent any one request or user overwhelming the system; using local or cloud-based filters to exclude excessively large requests or large numbers of requests; and using elastic infrastructure to allow you to temporarily and rapidly add capacity to deal with an overload condition.

It is worth noting that there is a fine line between an overload caused by legitimate (and thus profitable) traffic and that caused by a malicious actor who is deliberately trying to overload the system and create a denial-of-service attack. Telling the two apart can be very difficult in the early stages of a denial-of-service attack, and as we mention in [Chapter 7](#), using a commercial denial-of-service protection service<sup>6</sup> can be the most effective way to deal with this situation given the experience and tools that such services have at their disposal.

<sup>6</sup> Akamai, Arbor Networks, and Cloudflare are examples of denial-of-service protection vendors.

Ransomware attacks involve infiltrating the network with some form of hostile and malicious software agent (a “worm”) that replicates itself across the network and then, at some point, quickly or after a period of time, wakes up and encrypts all of the storage devices it can find. It then usually

halts the computer it is on and displays a message demanding payment of a ransom for a key to decrypt the organization's data. In the last few years, there have been several instances of large organizations being totally crippled by ransomware attacks.

The problem with dealing with a ransomware attack is that, by the time the organization realizes it has a problem, it is probably too late to stop the attack, and so it becomes a damage limitation and recovery exercise. Most experts advise against paying the ransom to deter future attacks and because the victim has no guarantee that the attacker will be able or prepared to decrypt the data even if the ransom is paid.

The key tactics for mitigating ransomware attacks are to prevent them happening in the first place and to ensure that you have regular, tested, comprehensive backups of your systems in place (including infrastructure, such as the data within your authentication systems). The use of automation to create and configure your IT platform will also be valuable, provided that you have a backup of this mechanism. Backups and platform automation allow you to re-create your IT environment from scratch to eliminate the ransomware agent from it. Preventing these attacks involves keeping your network as secure as possible and constant end-user education about (and automated detection of) **phishing attacks**, a favorite **attack vector** for ransomware.

On balance, the TFX system probably isn't a high-profile target for a denial-of-service attack, but we do need to be aware of possible availability threats and remember that, in the social-media-driven world of today, an activist campaign (perhaps about trade in a particular commodity) could suddenly drive unwelcome traffic to our APIs and Web interfaces. We also need to be aware of possible ransomware attacks, given the kind of organization that is likely to use TFX, and the team needs to work with the information security group on mitigating them. System availability is something we need to consider as part of resilience (which we explore in [Chapter 7](#)).

## Security Monitoring

One of the differences we've observed between security specialists and software engineers is their approach to security incidents. Software

engineers generally hope they won't happen, whereas security specialists know that they absolutely will. As we discussed earlier in this chapter, today's threat landscape means that most systems are attacked at some point; consequently, security monitoring is more important than ever.

If you are an architect or developer, then, in all likelihood, you won't be directly involved in operational security monitoring, which will probably be performed by a corporate security operations center (SOC) or by a third-party managed security services provider (MSSP) that you employ. Your role is to create a system that is monitorable and to work with those providing the operational monitoring to address incidents and problems if (when!) they occur.

In the context of TFX, we need to work early and often with the security operations organization within our parent company to get their operational perspective (part of principle 5, *Architect for build, test, deploy, and operate*). They almost certainly will want to include TFX in their overall threat monitoring and response scope, and we must make sure it is possible for them to do so.

Creating a system that can be monitored involves ensuring that key business operations are audited and system operations are logged. The audit trail allows a monitoring system to track what is happening within the system from a functional perspective, and logging allows it to track what is happening within the system from a technical perspective. We already mentioned that a robust audit trail is needed in TFX, and this is another reason why it is important.

In addition to auditing, a system needs to provide reliable, consistent, and meaningful message logging. This involves

- Logging startup and shutdown of system components.
- All executions of sensitive administrative- or security-related functions (e.g., payment authorizations, user authorization changes, failed password validations).
- Actions performed with special system access (e.g., during incident resolution).
- Runtime errors encountered by the system, abnormal requests received (e.g., malformed request payloads or unimplemented HTTP verbs).

- Any events relating to the logging system itself (e.g., it being stopped or files being rotated or archived).

The auditing and logging will undoubtedly evolve as the system grows and you gain more experience of it. The first penetration test will almost certainly reveal areas that need more logging and perhaps will reveal missing auditing too. Over time, however, working with the monitoring organization, you will be able to develop auditing and monitoring that rapidly allows anomalies to be spotted and investigated for possible security problems.

## Secrets Management

A security problem that is as old as security itself is where you keep the keys. Whether you are securing doors, safes, or encrypted data, you need to have a key (also known as a *secret* in the digital world) to unlock the protection and access the sensitive resource you want to access.

Historically, it is fair to say that secret storage was often quite careless, with secrets stored in plain text configuration files, database tables, operating system scripts, source code, and other insecure places. Today, we can't get away with this practice given the risk of compromise from both hostile actors and insiders.

This whole area is known as *secrets management*, and we must be very aware of it when designing our systems. Poor secrets management can render all of the rest of our security mechanisms useless. There are three main parts of secrets management: choosing good secrets, keeping them secret, and changing them reliably.

Most architects today are probably quite aware of the need for choosing good secrets. Secrets, which are keys such as passwords, encryption keys, or API keys, need to be long enough and random enough to be difficult to guess by brute force or intuition. There are many password generators<sup>7</sup> that can provide you with good keys, and most application platforms, such as Java and .NET, include secure key and random number generators. Secrets such as certificates and public key infrastructure (PKI) keys are normally generated automatically by reliable, proven utilities, such as `ssh-keygen`, on Unix systems.

<sup>7</sup> Such as Gibson Research Corporation's Perfect Passwords. <https://www.grc.com/passwords.htm>

Once you have chosen or generated secrets, those secrets, as their name suggests, need to be kept secret, but many people and many software applications legitimately require access to secrets, but of course, to a very specific and limited set of each. The best current approach to solving this problem is to use a dedicated and proven secrets manager. A secrets manager is a piece of software that safely stores encrypted versions of security secrets and restricts and audits access to them, using fine-grained security, via APIs and human interfaces. Ideally, a secrets manager should also provide some way of achieving high availability in case of failure of the secrets manager or its environment. Some of them also provide features to help with key rotation (discussed shortly).

All public cloud providers have a secret management service, and these tend to be robust, sophisticated, secure, and easy to use, so public cloud environments are often a good default choice. Where more control is needed, specific features are unavailable in a cloud service, or you need to store secrets yourself, there are a number of proven open source secrets managers, including Vault,<sup>8</sup> Keywhiz,<sup>9</sup> and Knox.<sup>10</sup>

<sup>8</sup> <https://www.vaultproject.io>

<sup>9</sup> <https://square.github.io/keywhiz>

<sup>10</sup> <https://github.com/pinterest/knox>

There are also many commercial products offering sophisticated secrets management, many supporting the key management interoperability protocol (KMIP) open standard<sup>11</sup> for interoperability, as well as more specialist solutions such as hardware security modules (HSMs), which can provide hardware-level secret storage. We don't have space here to discuss these more sophisticated solutions, but a security engineering book (such as Ross Anderson's, which we reference under "[Further Reading](#)") is a good place to find out more.

<sup>11</sup> <https://wiki.oasis-open.org/kmip>

The remaining part of secrets management is whether and how we change secrets regularly, often known as *key rotation*. There is a historical

orthodoxy that all secrets should be changed regularly, but in fact, if we think about the threats and risks that this practice mitigates, it quickly becomes clear that, for many secrets, it is of limited value (unless the secret is stolen, in which case it immediately needs to be changed).

For user passwords, the current view of many experts, including NIST<sup>12</sup> since 2017, is that you shouldn't force passwords to be changed unless they are compromised, as doing so does not improve authentication security. Similarly, for secrets such as database passwords and API keys, the only real benefit of regular changes is that if the secret is stolen without your knowledge, it will have a limited lifetime.

<sup>12</sup> Paul Grassi et al, *NIST Special Publication 800-63B, Digital Identity Guidelines, Authentication and Lifecycle Management* (National Institute of Standards and Technology, 2017).

In reality, though, unless you are using one-time passwords, almost any time from the secret being stolen to the password being changed will be long enough for adversaries to cause significant damage (and perhaps change the password to one that only they know). That said, if a breach occurs, you need to change passwords immediately, and in some cases, changing them regularly may be valuable. So, part of our secrets management approach needs to provide a robust mechanism for changing passwords.

Most of the secrets managers provide key rotation facilities for commonly occurring technologies, such as databases and operating systems, and you can often extend them to perform key rotation for your own technologies with plugins. Where this is impractical, there needs to be a well-understood and routinely practiced procedure, which is as automated as possible, to select a secure password, change the old password, and store the new one in the secret store.

## Social Engineering Mitigation

An old security maxim states that “you are only as secure as your weakest link,” and when many successful attacks are analyzed, the weakest links are not missing technical controls but the humans interacting with the system. This was even the case for the famous attack by an anonymous hacker on the Italian information security company HackingTeam in 2015.<sup>13</sup> If

HackingTeam can be totally compromised via social engineering and weak passwords, your organization probably can be too.

<sup>13</sup> J. M. Porup, “How Hacking Team Got Hacked,” *Ars Technica* (April 2016).

<https://arstechnica.com/information-technology/2016/04/how-hacking-team-got-hacked-phineas-phisher>

The problem with social engineering attacks, such as spoof phone calls, bribing employees, blackmail, and so on, is that we are dealing with the behavior of human beings who are, by nature, emotional and inconsistent, and so it is difficult to predict how they will behave. The best defense against reducing the probability of successful social engineering attacks is constant user education, reminders, and testing. The best defense against the cost of a social engineering attack is to minimize the impact that each user can have if compromised. There are a range of tactics that, as architects, we can use to help with both sorts of mitigation.

Our security mechanisms should always consider the possibility of subversion via social engineering. For example, we should consider protecting logins using 2FA to mitigate against attacks that attempt to steal passwords, particularly for sensitive logins such as administration of our secrets store or cloud infrastructure. We should actively limit the access that each user has in the system to the level of access required for their role, so that in the case of a social engineering attack, the user cannot inadvertently perform any action or access any data item in the system. Sensitive operations involving security or large financial transfers should be candidates for “four-eyes” controls, so that two people must agree to them. As discussed earlier, auditing should be widespread and visible so that people know that they are accountable for their actions. We should not embed links in email and expect people to click on them without clearly showing them what they are clicking on. And so on.

The security community is still learning how best to understand, model, and mitigate social engineering threats, but we can help protect our users against threats by considering human behavior when implementing security controls in our systems. For the TFX system, we’ve described most of the mitigations that the team can deploy, particularly a good audit trail (so that people know their actions are being recorded) and 2FA to make compromising accounts via stolen passwords more difficult. They also

should consider some multiuser (four-eyes) controls for sensitive operations (perhaps large payment authorizations).

## Zero Trust Networks

Our historical approach to securing systems is to use *zones of trust* from the outside to the inside, with the level of trust increasing as we enter the network. A standard pattern for zones of trust is a typical enterprise network with an Internet zone (the area outside the firm), a trusted zone (the firm's own private network), and between the two a demilitarized zone, or DMZ, which is partially trusted but isolated from the trusted main corporate network. A privileged zone often exists within the trusted zone, and it holds particularly sensitive information, such as credit card data. To isolate the zones from each other, a complex set of network routing and firewall rules is used to limit the traffic that can flow between them.

This approach served the industry well for the first 10 to 15 years of Internet computing, but its limitations have started to become clear in recent years, with the ever-increasing sophistication of attackers and the resulting number of high-profile, extremely sensitive data breaches from large organizations. The main limitation of the traditional approach is that once one of the zones is breached by an adversary, there is little to stop the adversary from accessing anything within it. Given some time, skill, and determination, an attacker can probably access the entire zone, because the security controls assume that everything in the zone can be trusted to access everything else within it. Hence, a compromised administrator's credential can often lead to a disastrous data breach.

Given the evolution of the threat landscape that we discussed earlier, an alternative approach has been gaining in popularity in recent years, known as *zero trust security*, with roots in ideas such as Google's BeyondCorp model<sup>14</sup> and the book *Zero Trust Networks*<sup>15</sup> by Barth and Gilman.

<sup>14</sup> Rory Ward and Betsy Beyer, "BeyondCorp: A New Approach to Enterprise Security," *login*: 39, no. 6 (2014): 6–11. <https://research.google/pubs/pub43231>

<sup>15</sup> Evan Gilman and Doug Barth, *Zero Trust Networks* (O'Reilly Media, 2017).

A true zero trust environment is a sophisticated and complex infrastructure-led undertaking, which, as a reader of this book, you are probably involved

in but are unlikely to be leading. However, our role as software architects is to be aware of the approach, understand its benefits and costs, and apply its principles where relevant to our systems. In particular, the idea that we should explicitly validate our trust relationships and assume that the network we are using is hostile is an excellent security principle to bear in mind. We should be designing our systems with the assumptions that an adversary already has access to parts of our infrastructure, that our network is open to interception, and that a degree of paranoia is healthy and normal. If we take this approach in the security design of our systems, we are much more likely to be resilient to insider and hostile adversary threats, even if our network is quite secure.

## Achieving Security for TFX

We don't have enough space to explore every aspect of securing the TFX system, but we show how to approach one representative part of the process, illustrating how to use the information in this chapter to ensure the security of the systems that you build.

We make some assumptions to help us focus on one of the more interesting aspects of TFX's security needs. First, we assume that in addressing some basic threats, our team has implemented some relatively common security mechanisms:

- The team has incorporated authentication and authorization, using an application framework<sup>16</sup> and a cloud-based authentication and authorization provider such as Okta, to allow identification and authorization of system users.

<sup>16</sup> Such as Spring Security. <https://spring.io/projects/spring-security>

- The team has considered interception threats and consequently implemented transport-level security (TLS) between the system's services.
- The team has considered spoofing, denial of service, and elevation of privilege attacks, and so TFX has network security controls, preventing network traffic from outside the system's network from accessing any part of TFX apart from the API gateway. We discuss the

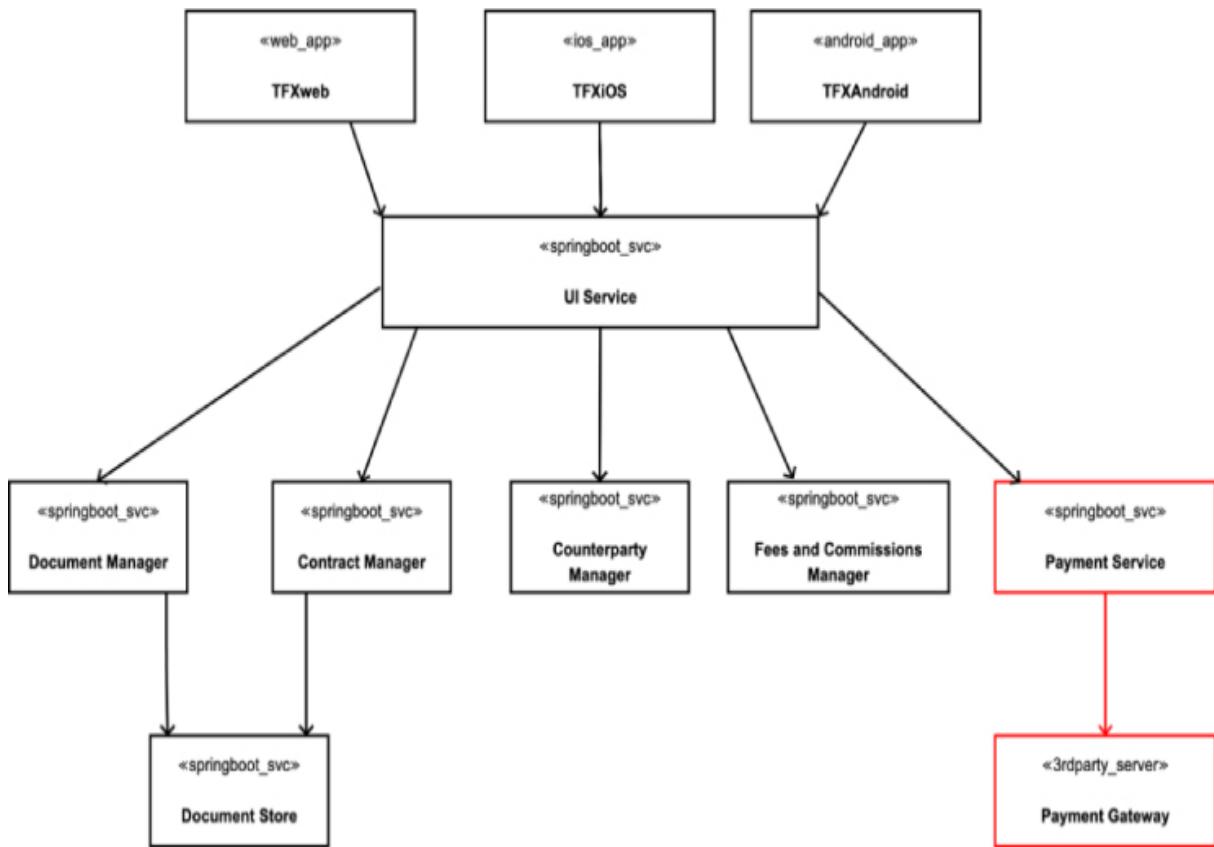
API gateway in more detail in [Chapter 5](#), “Scalability as an Architectural Concern.”

- Again, to avoid spoofing threats, TFX has authorization controls in the API gateway so that only users with authorized access to TFX can invoke API calls on its services.
- Credentials (such as those for the databases) are kept in the secrets store, available as a service within the TFX cloud environment.

The team has made a good start at securing TFX and have mitigated some of the common threats TFX is likely to face. What is next? Let’s consider our first threat, which we identified in [Table 4.1](#) as TFX-123, namely, the danger of an adversary with access to the network sending an unauthorized payment request in an attempt to transfer funds illicitly. How would this threat be mitigated? (It is primarily a spoofing threat, as it involves falsifying the identity of the sender of the request and the details within the request.)

When we look at the most relevant parts of TFX—the Payment Service and Payment Gateway, as highlighted in [Figure 4.6](#)—the threat is that an attacker somehow manages to get the Payment Gateway to process a payment request.

The full notation we are using is explained in [Appendix A](#), but for [Figure 4.6](#), the rectangular symbols are system components, with the type of component indicated in the guillemot quotes («»). The arrows between the components indicate intercomponent communication (with the arrowhead indicating the direction of initiation of the communication). We have shaded the parts of [Figure 4.6](#) that we are interested in for this example.



**Figure 4.6 Securing TFX payments**

As the attacker has access to the network, the attack vectors they are likely to use are to send a request to the Payment Service to fool it into creating the payment request through the Payment Gateway or to send a malicious request directly to the Payment Gateway to trigger the payment.

Given the security mechanisms (listed previously) that the team has put in place to mitigate other threats, their task is to make sure that payment requests to either service can originate only from one of our services and not from an attacker injecting requests into the network. This is a simple example of the zero trust principle outlined earlier, as we are assuming that we cannot trust the local network.

There are two common mechanisms they can use to ensure against **injection attack**. First, they could use API authentication via the authentication and authorization service that they integrated into our programming framework,<sup>17</sup> and second, they could use client certificates on the TLS connections that are already in place. In this situation, as we do not

trust the network and have limited control over the Payment Gateway software, they need to do both.

<sup>17</sup> Which in our example is Spring Security.

While it is possible to modify the Payment Service to call an external authentication mechanism, the Payment Gateway is a third-party software component, and it turns out that this product does not provide easy integration with an external authentication service, even if the authentication service uses open standards (e.g., Security Assertion Markup Language [SAML] 2.0 at the time of writing). The Payment Gateway also needs to run on its own server (for resource isolation reasons, discussed in [Chapter 7](#)). This combination of factors is why we need to both use API security and secure the network connection. Therefore, the team implements an authorization check in the Payment Service, configures the Payment Gateway to accept only inbound requests across TLS, and configures the TLS endpoint to accept only requests signed by the Payment Service's primary key.

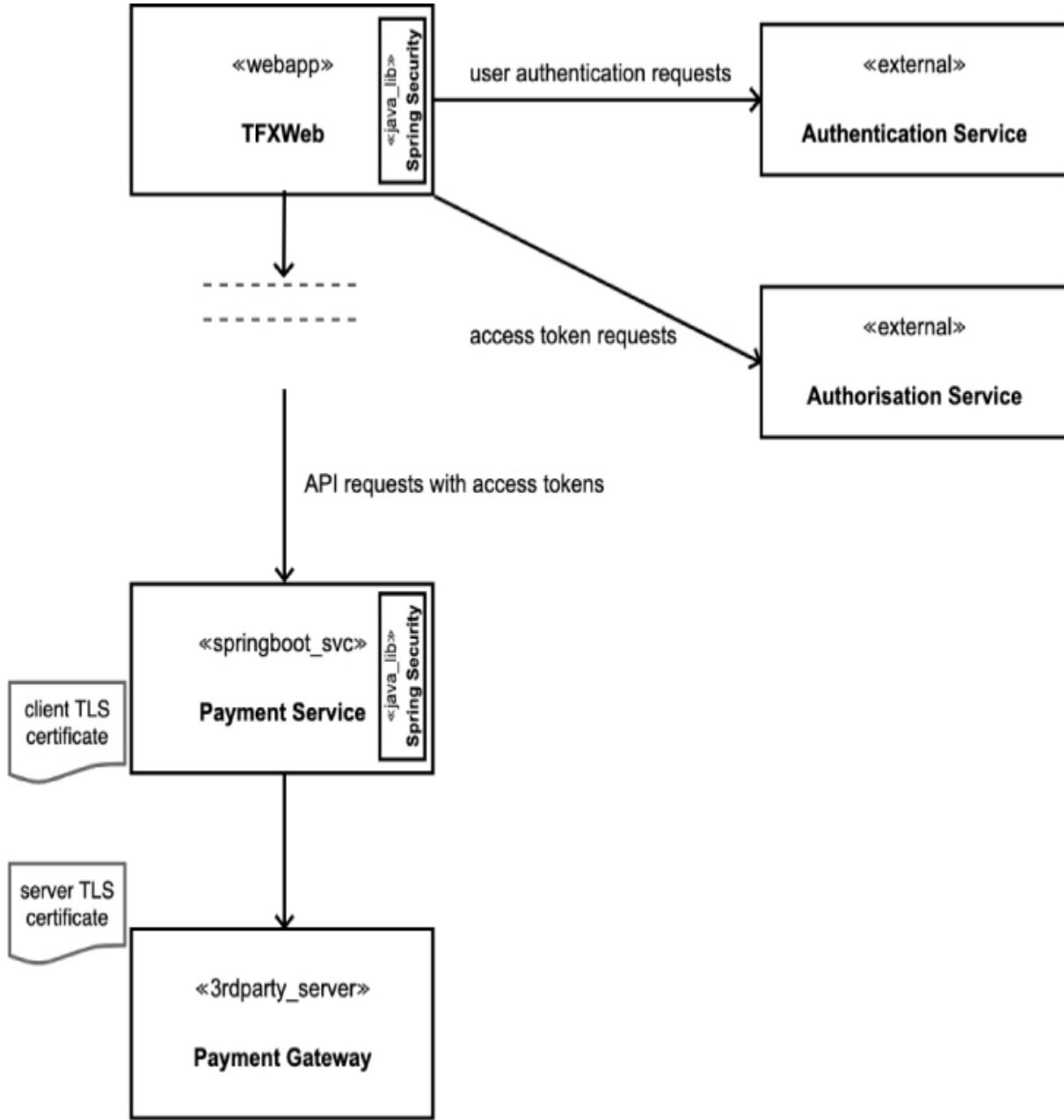
This sort of complication is quite common when securing real systems. Securing the parts that we control can be a lot simpler than securing a combination of our own and third-party software components, some of which may be quite old.

The team could also add network-level controls to limit network traffic so that only traffic from the Payment Service can be routed to the Payment Gateway, but doing so adds further complexity to the TFX environment, as it involves installing and maintaining additional network devices or software firewall configuration on the servers. The team decide to avoid this mechanism at this stage, but of course, a security review or further threat analysis might indicate the need to revisit this option later.

The informal diagram in [Figure 4.7](#) shows how the team augment the architectural design to implement these controls.

The notation in [Figure 4.7](#) is similar to the notation used in [Figure 4.6](#). The rectangles are system components, annotated with their types, and the arrows are the connections between the components. The figure also shows specific programming libraries embedded in several of the components, to

make their use clear. The rectangles with curved bases indicate where specific cryptographic (TLS) certificates are needed.



**Figure 4.7** *Payment Gateway security*

As shown in [Figure 4.7](#), TFXWeb is extended to call an external authentication service to identify the end user, and before calling any of the APIs, the Web application calls an authorization service to request an access token<sup>18</sup> that securely encodes the rights of that user. The token is then

passed as part of the API requests (in an HTTP header) for the services to use. The Payment Service verifies the structure and signature of the token (ideally using a proven existing solution, e.g., Spring Security, rather than custom code) to ensure it can be trusted and then uses the information within it to allow or deny the request. As we said, the Payment Gateway is third-party software, so we can't implement a similar check within it. The team therefore configure a bidirectional TLS connection between it and the Payment Service. The Payment Gateway is configured to accept requests only from a specific client-side certificate that they generate and install in the Payment Service, and in turn, the server-side certificate is used to ensure that the Payment Service is connecting to the Payment Gateway.

<sup>18</sup> Such as a JSON Web Token (JWT) format token.

We have added some complexity to our system, and possibly operational cost to maintain the security mechanisms, but unfortunately, some complexity and cost are inevitable if we want effective security.

## Maintaining Security

Having a system that is designed to be secure is an important first step in having a system that actually is secure in practice. The system then needs to be built securely and must remain secure in the real world where it operates. This section discusses the associated concerns that we need to be aware of as architects.

### Secure Implementation

Achieving a secure system involves work in the areas of security requirements, security design, secure implementation, and security testing, and all four parts are equally important. There is clearly no point in carefully designing a secure system if we then implement it carelessly and thereby introduce security flaws or if we fail to test the security features of the system thoroughly.

The topics of secure implementation and security testing are large enough to fill an entire book by themselves. If you don't already have knowledge of and experience with secure development and security testing, we encourage

you to read some of the references listed under “[Further Reading](#)” to deepen your knowledge in this area and to use the resources of organizations such as OWASP<sup>19</sup> and SAFECode<sup>20</sup> to ensure that your system is implemented and tested just as securely as it has been designed.

<sup>19</sup> Open Web Application Security Project. <https://www.owasp.org>

<sup>20</sup> <https://safecode.org>

## People, Process, Technology

Technologies such as encryption, authentication, firewalls, and auditing are important, but as security-aware engineers and architects, we must remember that there are three equally important aspects to security: people, process, and technology.

As previously mentioned, people are a crucial part of any secure system. It is often the awareness, knowledge, and behavior of a system’s users and administrators that results in much of its effective day-to-day security. A simple mistake, such as an administrator writing down a password on a sticky note or a user clicking a phishing attack link in an email, can negate any amount of security technology in the system.

Similarly, most systems have a certain amount of operational process wrapped around them to make sure that they operate smoothly and are resilient in the face of unexpected situations. It is common to see a lot of effort go into operational procedures to ensure that they are comprehensive, as simple as possible, and flexible in the face of unexpected situations, yet their security implications often are not thoroughly considered. Some of the classic operational mistakes are not testing the configuration of newly commissioned infrastructure items for security, deferring or forgetting the patching of operating systems and system software, backing up secure data stores regularly but failing to protect the backups effectively, and backing up data carefully but failing to test that the restore process works.

Finally, we need to apply security technology to our system to mitigate its security threats. We must use the simplest, most reliable, and best-understood technology that will do the job, even though this often runs against our instincts to use the newest and most interesting technology for the job!

## The Weakest Link

As mentioned earlier, you are only as secure as your weakest link, and a few moments of logical thought confirms that principle as a tautology. Of course, by definition, the part of the system with the weakest security defines the security level of the system.

Reality is slightly more nuanced than this because different parts of a system may be exposed to different levels of security threat, and the severity of a security breach is likely to be different for different parts of the system. However, the basic point is still valid.

The reason it is important to keep the weakest link in mind is that it reminds us that security is not a binary state. Like other quality attributes, there are degrees of security, and our current level is defined by the part of the system with the weakest security defenses relative to the threat faced.

The challenge in turning this principle into specific action is that our weakest security link often is not obvious. Therefore, we should use this principle as a reminder to keep probing our system for weak links (security testing) and keep threat modeling throughout the life of the system, continually improving the security where we find it lacking. Which brings us neatly on to our next point.

## Delivering Security Continuously

Historically, security, like some other quality attributes, has been addressed at the end of the life cycle and the work done once to “secure” the system before its release. This late consideration of security and associated late involvement of the security group inevitably leads to unexpected surprises, delays, and the security team being perceived as the group who always says no.

While this big-bang, late-in-the-day approach may have been acceptable for biannual, high-ceremony releases, it doesn’t work for our modern cloud-first, continuous delivery world. We need to rethink how we integrate security into our delivery life cycle.

Today, we need to view security as a continuous activity that starts as the system is being conceived, continues through the development process, and

is ever-present in operation. In other words, we need continuous security as part of continuous delivery.

We achieve this by doing small amounts of security work during every sprint or iteration and integrating security tests and controls into our continuous delivery pipelines. This approach allows us to treat security as an incremental process of identifying the new security risks as they emerge, learning from our experience of operating the system in production, and continually mitigating the threats that we identify.

The need for this approach to application security is one of the reasons for the emergence of the DevSecOps movement, extending DevOps to include security as an integrated concern throughout the system delivery life cycle.

As we have said throughout this chapter, in our TFX system, we will start out with robust but simple security facilities to address our highest risk threats and allow us to safely move the system to operation. As we add more features and better understand our risks, we'll refine and add more security, sprint by sprint. In one sprint, we might want to start penetration testing and add encryption of some documents, as we're moving past the pilot phase and now have genuinely sensitive documents in the system. A few sprints later, we might implement the audit trail we discussed earlier, as we're starting to see real usage of potentially sensitive activities, and engage a third party to perform continual simulated security attacks to check our defenses and ability to respond. And a little later, as we start to see significant financial transactions in our system, we'll add the requirement for two-user reviews (four-eyes checks) for significant operations, such as releasing payments above a certain amount.

By working in this incremental way, we have remembered principle 3, *Delay design decisions until they are absolutely necessary*, and have made security an enabler for the cloud-first, continuous delivery world rather than an obstacle to be worked around.

## Being Ready for the Inevitable Failure

As any experienced software architect or engineer will tell you, once systems are deployed and used by real users, things go wrong and often in rather unexpected ways. Unfortunately, this is as true for security as for any other aspect of the system, such as functionality or scalability. The

difference is that certain sorts of security failures can lead to events, such as large-scale consumer data exposure, that have lasting impact on an organization's reputation and ability to trade.

The increasingly hostile threat landscape that we explained at the start of this chapter means that, quite apart from any mistakes we make with our security implementation, any system of any value is likely to be attacked, quite possibly by skilled adversaries. At some point, then, we are likely to face some sort of security failure.

Consequently, we must design our systems with recovery from security incidents in our mind and must think about preventing the incidents in the first place. We discuss system resilience in [Chapter 7](#), but from a security perspective, an important part of ensuring resilience is to work closely with the security specialists and SOC personnel while designing a system so that we understand their needs for incident recognition and response. Consider involving the development team in the work of the SOC and the security specialists, perhaps using a rotation model. Also make sure that development team members are involved in security retrospectives after incidents or near misses. This sort of cooperation provides the development team with insights into the work of the security teams and greatly increases their security awareness.

The other aspect of being ready for security problems is to think through security failures when considering the recovery mechanisms for a system. What if we had to re-create our TFX system from scratch in a new cloud environment? What if our certificates were compromised? What if we were the subject of a ransomware attack? What if our authorization system was compromised or unavailable? What if we discovered that the system had been compromised and data had been exfiltrated already? By thinking through such unpleasant possible scenarios, we can design a system that can be recovered from attack and is as resilient as possible against such threats.

## Security Theater versus Achieving Security

In most large organizations, many aspects of security practice and processes are simply performed with no one seeming to understand why. The tasks might be the result of a single problem that occurred some years ago (“organizational scar tissue”), activities that were once useful but are no

longer relevant, or perhaps important parts of the organization's security defenses, but the reason for their continued practice has been lost.

The problem is that if we have a lot of these activities that are not understood, we don't know which are effective and important and which are just "security theater," as Bruce Schneier<sup>21</sup> terms it, meaning security activities that are very visible but have no real impact on the organization's security level.

<sup>21</sup> Bruce Schneier, "Beyond Security Theater," *New Internationalist* (November 2009). [https://www.schneier.com/essays/archives/2009/11/beyond\\_security\\_thea.html](https://www.schneier.com/essays/archives/2009/11/beyond_security_thea.html)

As software engineers and architects our job is to try to use our technical and logical thinking skills to identify which is which and to challenge those that are not useful. Which activities help us to really be secure and which are outdated or simply useless "security theatre"? If we don't do this then all security activities will be devalued and none of them will be taken seriously. So massively weakening the organization's resilience to security threats.

## Summary

Gone are the days when software architects could consider security to be largely someone else's problem and could be handled by adding authorization and authentication to a system. Today's threat landscape and ubiquitous network connections mean that our system is as likely to be attacked as is a traditional target, such as a payments company.

Security work needs to be integrated into our architecture work as a routine part of the continuous architecture cycle, with proven techniques from the security community, such as threat modeling, used to identify and understand threats, before we attempt to mitigate them using security mechanisms and improved processes and training our people.

However, security work also has to evolve given the need today to move quickly and continually and the increasing sophistication of the threat landscape. It is no longer useful or effective to schedule multiday security reviews twice a year and produce pages and pages of findings for manual remediation. Today, security needs to be a continual, risk-driven process

that is automated as much as possible and supports rather than blocks the continuous delivery process.

By working in this cooperative, risk-driven, continuous way, software architects and security specialists can ensure that their systems are ready and continue to be ready to face the security threats that they are inevitably going to face.

## Further Reading

This section includes a list of resources that we found helpful to further our knowledge in the topics that we have touched on in this chapter.

- For a broad introduction to modern application-oriented security practice, *Agile Application Security* (O'Reilly Media, 2017) by Laura Bell, Michael Brunton-Spall, Rich Smith, and Jim Bird is a good place to start. This book explains how to integrate sound, modern security practice into an agile development life cycle. It is based on the author team's experience doing exactly that for parts of the UK government. An older book, soon to appear in a new third edition, is Ross Anderson's classic *Security Engineering*, 3rd ed. (Wiley, 2020), which provides a very broad, accessible, entertaining, and thorough introduction to the field. The previous edition is freely available in PDF.
- A good Web-based resource to help you engage development teams in software security is Charles Weir's Secure Development,<sup>22</sup> which contains the materials for an approach called Developer Security Essentials, an accessible and inclusive approach to getting development teams involved in security.
- There are many threat modeling approaches to choose from, but in our opinion, a good place to start is with Adam Shostack's *Threat Modeling: Designing for Security* (Wiley, 2014), based on his experience introducing threat modeling to Microsoft's product development organization. It primarily talks about STRIDE but has a chapter on attack trees too, although Shostack is not particularly enthusiastic about them. The original reference on attack trees is Bruce Schneier's "Attack Trees" (*Dr. Dobb's Journal*, July 22, 2001),<sup>23</sup>

although you will find plenty of other material about them using an Internet search.

<sup>22</sup> <https://www.securedevelopment.org>

<sup>23</sup> [https://www.schneier.com/academic/archives/1999/12/attack\\_trees.html](https://www.schneier.com/academic/archives/1999/12/attack_trees.html)

Each of the specific threat modeling approaches has its own documentation:

PASTA—Marco Morana and Tony Uceda Vélez, *Risk Centric Threat Modeling: Process for Attack Simulation and Threat Analysis* (Wiley, 2015)<sup>24</sup>

VAST—Alex Bauert, Archie Agarwal, Stuart Winter-Tear, and Dennis Seboyan, “Process Flow Diagrams vs Data Flow Diagrams in the Modern Threat Modeling Arena” (2020)<sup>25</sup>

OCTAVE—Christopher Alberts, Audrey Dorofee, James Stevens, and Carol Woody, *Introduction to the OCTAVE Approach* (Software Engineering Institute, Carnegie Mellon University, 2003)

MITRE’s CAPEC attack catalog—<https://capec.mitre.org>

NIST approach for data-intensive systems—Murugiah Souppaya and Karen Scarfone, *Guide to Data-Centric System Threat Modeling*, NIST Special Publication 800-154 (NIST, 2016)

The Software Engineering Institute (SEI)’s cybersecurity researchers have also written a report summarizing some of these approaches: Nataliya Shevchenko et al., *Threat Modeling: A Summary of Available Methods* (Software Engineering Institute, Carnegie Mellon University, 2018).

Threat modeling is a type of risk management, and in some organizations, it sits within a broader security risk analysis and mitigation process, such as the SEI’s Security Engineering Risk Analysis (SERA) method, which may be valuable if a broader and more formal approach to cybersecurity risk management is needed. See Christopher Alberts, Carol Woody, and Audrey Dorofee, *Introduction to*

*the Security Engineering Risk Analysis (SERA) Framework* (CMU/SEI-2014-TN-025) (Software Engineering Institute, Carnegie Mellon University, 2014).

- Social engineering is a broad area but one that we have found relatively little in-depth material on, however, *Social Engineering: The Art of Human Hacking*, 2nd ed. (Wiley, 2018) by Christopher Hadnagy explores this complex topic quite thoroughly.
- Finally, two organizations we mentioned earlier are SAFECode and OWASP. They both have produced a lot of valuable information for improving application development security, all of which is freely available. Notable resources are the SAFECode’s “Fundamental Practices for Secure Software Development, Third Edition” (2019);<sup>26</sup> the OWASP “cheat sheets,” such as the “Key Management Cheat Sheet” (2020)<sup>27</sup> and the well-known OWASP “Top 10 Threats” lists for web applications (2017),<sup>28</sup> for mobile applications (2016),<sup>29</sup> and for APIs (2019).<sup>30</sup>

<sup>24</sup> An Internet search will also return a number of whitepapers and presentations on the PASTA technique.

<sup>25</sup> <https://go.threatmodeler.com/process-flow-diagrams-vs-data-flow-diagrams-in-threat-modeling>

<sup>26</sup> <https://safecode.org/fundamental-practices-secure-software-development>

<sup>27</sup> [https://cheatsheetseries.owasp.org/cheatsheets/Key\\_Management\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Key_Management_Cheat_Sheet.html)

<sup>28</sup> [https://owasp.org/www-project-top-ten/OWASP\\_Top\\_Ten\\_2017](https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017)

<sup>29</sup> <https://owasp.org/www-project-mobile-top-10>

<sup>30</sup> <https://owasp.org/www-project-api-security>

# Chapter 5. Scalability as an Architectural Concern

*Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning.*

— Albert Einstein

Scalability has not been traditionally at the top of the list of quality attributes for an architecture. For example, *Software Architecture in Practice*<sup>1</sup> lists availability, interoperability, modifiability, performance, security, testability, and usability as prime quality attributes (each of those has a dedicated chapter in their book) but mentions scalability under “Other Quality Attributes.” However, this view of scalability as relatively minor has changed during the last few years, perhaps because large Internet-based companies such as Google, Amazon, and Netflix have been focusing on scalability. Software architects and engineers realize that treating scalability as an afterthought may have serious consequences should an application workload suddenly exceed expectations. When this happens, they are forced to make decisions rapidly, often with insufficient data. It is much better to take scalability into serious consideration from the start. But what exactly do we mean by scalability in architecture terms, and what is its importance?

<sup>1</sup> Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice*, 3rd ed. (Addison-Wesley, 2012).

Scalability can be defined as the property of a system to handle an increased (or decreased) workload by increasing (or decreasing) the cost of the system. The cost relationship may be flat, may have steps, may be linear, or may be exponential. In this definition, *increased workload* could include higher transaction volumes or a greater number of users. A *system* is usually defined as a combination of software and computing infrastructure and may include human resources required to operate the software and associated infrastructure.

Going back to the trade finance case study, the Trade Finance eXchange (TFX) team understands that creating a scalable TFX platform will be an important success factor. As explained in [Chapter 1](#), “[Why Software Architecture Is More Important than Ever](#),” the initial aim is to offer the letters of credit (L/C) platform that the team is developing to a single financial institution. In the longer term, the plan is to offer it as a white label<sup>2</sup> solution for other financial institutions. Our challenge in creating a scalable platform is that we don’t really have good estimates of the associated increases (and rate of increase) in terms of transaction volumes, data size, and number of users. In addition, the team is struggling to create a scalable architecture when using the Continuous Architecture principles and toolbox. Principle 3, *Delay design decisions until they are absolutely necessary*, directs the team not to design for the unknown, and scalability estimates are unknown at this point. How the team can successfully overcome the challenge of not knowing their future state is described in this chapter.

<sup>2</sup> A white label product is a product that can be branded by our customers.

This chapter examines scalability in the context of Continuous Architecture, discussing scalability as an architectural concern as well as how to architect for scalability. Using the TFX case study, it provides practical examples and actionable advice. It is important to note that, although we focus on the architectural aspects of scalability, a complete perspective for scalability requires us to look at business, social, and process considerations as well.<sup>3</sup>

<sup>3</sup> For more details on this topic, see Martin L. Abbott and Michael J. Fisher, *The Art of Scalability* (Addison-Wesley 2015).

## Scalability in the Architectural Context

To understand the importance of scalability, consider companies such as Google, Amazon, Facebook, and Netflix. They handle massive volumes of requests every second and are able to successfully manage extremely large datasets in a distributed environment. Even better, their sites have the ability to rapidly scale without disruption in response to increased usage due to expected events such as the holiday season or unexpected events such as health incidents (e.g., the COVID-19 pandemic) and natural disasters.

Testing for scalability is generally difficult for any system, as the resources needed may be available only in the production environment, and generating a synthetic workload may require significant resources and careful design. This is especially difficult for sites that need to handle high and low traffic volumes without precise volume estimates. Yet companies such as Google, Amazon, Facebook, and Netflix excel at achieving scalability, and this is a critical factor behind their success. However, software architects and engineers need to remember that scalability tactics used at those companies do not necessarily apply to other companies that operate at a different scale.<sup>4</sup> They need to be careful about too quickly reusing scalability tactics in their own environments without thoroughly understanding the implications of the tactics they are considering and documenting their requirements and assumptions explicitly as a series of architectural decisions (refer to [Chapter 2, “Architecture in Practice: Essential Activities”](#)).

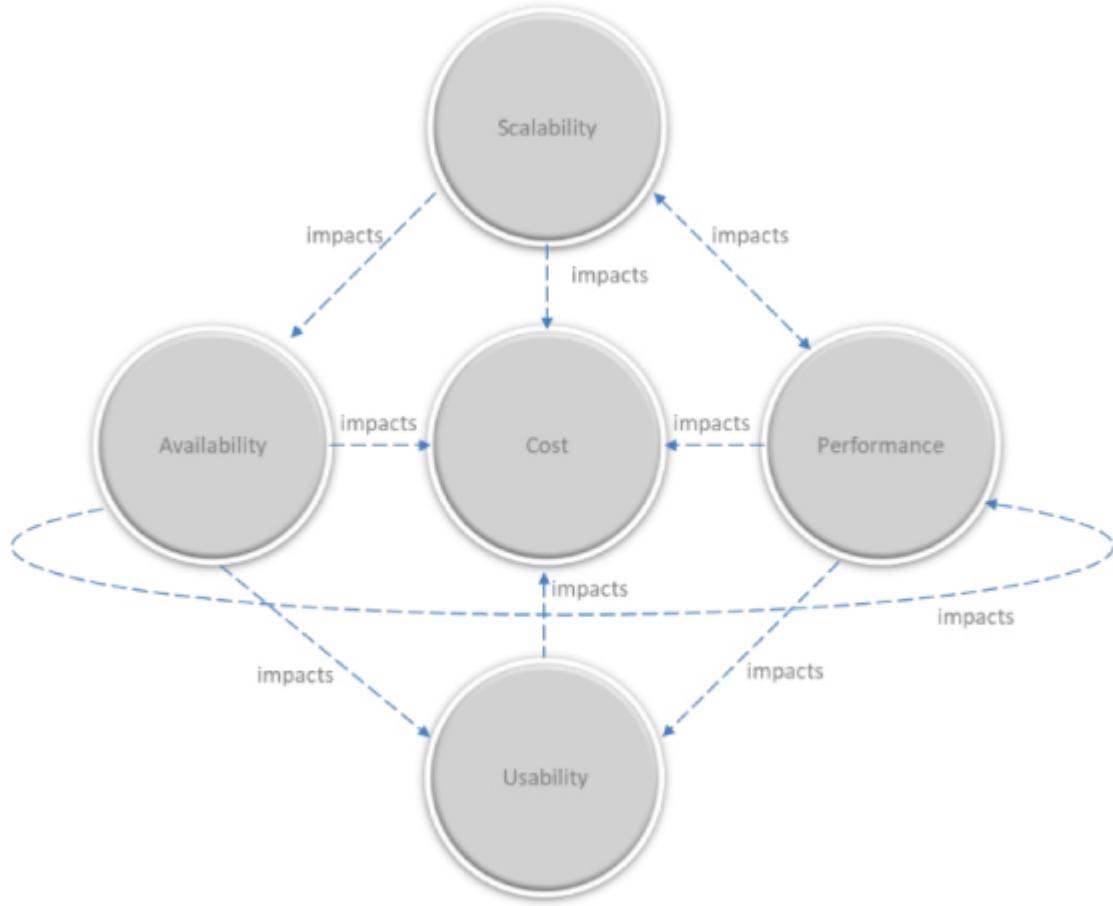
<sup>4</sup> Oz Nova, “You Are Not Google” (2017). <https://blog.bradfieldcs.com/you-are-not-google-84912cf44afb>

In [Chapter 2](#), we recommended capturing quality attribute requirements in terms of stimulus, response, and measurement. However, the TFX stakeholders are struggling to provide scalability estimates for TFX because it is difficult to predict the future load of a system, as a lot depends on how successful it is.

Stakeholders may not always be a good source for scalability requirements, but those requirements can often be inferred from business goals. For example, in order to achieve a target revenue goal, a certain number of transactions would need to be conducted, assuming a particular conversion rate. Each of these estimated factors would have its own statistical distribution, which might tell you that you need 10 times more standby capacity than average to be able to handle loads at the 99th percentile.

Given the lack of business objectives for TFX, the team decides to focus their efforts on other, better-documented quality attribute requirements, such as those for performance (see [Chapter 6, “Performance as an Architectural Concern”](#)) and availability (see [Chapter 7, “Resilience as an Architectural Concern”](#)). Unfortunately, the team quickly notices that

scalability impacts other quality attributes, as shown in [Figure 5.1](#). They clearly can't ignore scalability.



**Figure 5.1** Scalability–performance–availability–usability cost relationships

The team also realizes that the four quality attributes, especially scalability, have an impact on cost. As we explained in [chapter 3](#) of *Continuous Architecture*,<sup>5</sup> cost effectiveness is not commonly included in the list of quality attribute requirements for a system, yet it is almost always a factor that must be considered. Architects of buildings and ships, as well as designers of everything from aircraft to microchips, routinely propose several options to their customers based on cost. All clients are cost sensitive, regardless of the engineering domain.

<sup>5</sup> Murat Erder and Pierre Pureur, *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World* (Morgan Kaufmann, 2015).

Developing an executable architecture quickly and then evolving it are essential for modern applications in order to rapidly deliver software systems that meet ever-changing business requirements. However, you do not want to paint yourself into a corner by underestimating the economic impact of scalability. Not planning for scalability up front and hoping that your platform will be able to survive a sudden load increase by relying on a simple mechanism such as a load balancer<sup>6</sup> and leveraging an ever-increasing amount of cloud-based infrastructure may turn out to be an expensive proposition. This strategy may not even work in the long run.

<sup>6</sup> A load balancer is a network appliance that assigns incoming requests to a set of infrastructure resources in order to improve the overall processing.

Architectural decisions for scalability could have a significant impact on deployment costs and may require tradeoffs between scalability and other attributes, such as performance. For example, the TFX team may need to sacrifice some performance initially by implementing asynchronous interfaces between some services to ensure that TFX can cope with future loads. They may also increase the cost of TFX by including additional components such as a message bus to effectively implement interservice asynchronous interfaces. It is becoming clear that scalability is a key driver of the TFX architecture.

## What Changed: The Assumption of Scalability

Scalability was not a major concern decades ago, when applications were mainly monolithic computer programs running on large Unix servers, and comparatively few transactions would be operating close to real time, with almost no direct customer access. Volumes and rate of change, as well as costs of building and operating those systems, could be predicted with reasonable accuracy. Business stakeholders' expectations were lower than they are today, and the timeframe for software changes was measured in years and months rather than days, hours, and minutes. In addition, the businesses of the past were usually less scalable owing to physical constraints. For example, you could roughly estimate the daily transaction volume of a bank by the number of tellers the bank employed because all transactions were made through tellers. Today, unexpected transaction

volumes can arrive at any time, as there is no human buffer between the systems and the customers.

As we described in [Chapter 1](#), software systems evolved from monoliths to distributed software systems running on multiple distributed servers. The emergence and rapid acceptance of the Internet enabled the applications to be globally connected. The next evolution was the appearance and almost universal adoption of cloud computing, with its promise of providing a flexible, on-demand infrastructure at a predictable, usage-based optimal cost. Software architects and engineers quickly realized that even if monolithic software systems could be ported with minimal architecture changes to distributed environments and then to **cloud infrastructures**, they would run inefficiently, suffer scalability problems, and be expensive to operate. Monolithic architectures had to be refactored into service-oriented architectures and then into microservice-based architectures. Each architecture evolution required a system rewrite using new technologies, including new computer languages, middleware software, and data management software. As a result, software architects and engineers are now expected to learn new technologies almost constantly and to rapidly become proficient at using them.

The next evolution of software systems is the move toward intelligent connected systems. As described in [Chapter 8, “Software Architecture and Emerging Technologies,”](#) artificial intelligence (AI)-based technologies are fundamentally changing the capabilities of our software systems. In addition, modern applications now can directly interface with intelligent devices such as home sensors, wearables, industrial equipment, monitoring devices, and autonomous cars. For example, as described in [Chapter 3, “Data Architecture,”](#) one of the big bets for TFX is that it could alter the business model itself. Could the team replace the current reliance on delivery documentation with actual, accurate, and up-to-date shipping status of the goods? Sensors could be installed on containers used to ship the goods. These sensors would generate detailed data on the status of each shipment and the contents of the containers. This data could be integrated into and processed by TFX. We discuss this in more detail in the [“Using Asynchronous Communications for Scalability”](#) section.

Each evolution of software systems has increased their accessibility and therefore the load that they need to be able to handle. In a span of about 40

years, they evolved from software operating in batch mode, with few human beings able to access them and with narrow capabilities, into ubiquitous utilities that are indispensable to our daily lives. As a result, estimating transaction volumes, data volumes, and number of users has become extremely difficult. For example, according to the BigCommerce website, each month in 2018, more than 197 million people around the world used their devices to visit Amazon.com.<sup>7</sup> Even if those numbers could be predicted a few years ago, how could anyone predict the traffic volumes caused by extraordinary events such as health incidents (e.g., COVID-19 pandemic) when Amazon.com suddenly became one of the few stores still selling everyday goods during country and state lockdowns?

<sup>7</sup> Emily Dayton, “Amazon Statistics You Should Know: Opportunities to Make the Most of America’s Top Online Marketplace.” <https://www.bigcommerce.com/blog/amazon-statistics>

## Forces Affecting Scalability

One way to look at scalability is to use a supply-and-demand model frame of reference. Demand-related forces cause increases in workload, such as increases in number of users, user sessions, transactions and events. Supply-related forces relate to the number and organization of resources that are needed to meet increases in demand, such as the way technical components (e.g., memory, persistence, events, messaging, data) are handled. Supply-related forces also relate to the effectiveness of processes associated with the management and operation of the technical component as well as the architecture of the software.

Architectural decisions determine how these forces interact. Scaling is optimized when there is a balance between these forces. When demand-related forces overwhelm supply-related forces, the system performs poorly or fails, which has a cost in terms of customer experience and potentially market value. When supply-related forces overwhelm demand, the organization has overbought and needlessly increased its costs, although using a commercial cloud may minimize this issue.

## Types and Misunderstandings of Scalability

Calling a system scalable is a common oversimplification. Scalability is a multidimensional concept that needs to be qualified, as it may refer to

application scalability, data scalability, or infrastructure scalability, to name a few of many possible types. Unless all of the components of a system are able to cope with the increased workload, the system cannot be considered scalable. Assessing the scalability of TFX involves discussing scenarios: Will TFX be able to cope with an unexpected transaction volume increase of 100 percent over the estimates? Even if TFX can't scale, could it fail gracefully (many security exploits rely on hitting the system with unexpected load and then taking over when the application fails)?<sup>8</sup> Will the platform be able to support a significant number of customers beyond the initial client without any major changes to the architecture? Will the team be able to rapidly add computing resources at a reasonable cost if necessary?

<sup>8</sup> See [Chapter 4, “Security as an Architectural Concern,”](#) and [Chapter 7, “Resilience as an Architectural Concern.”](#)

Software architects and engineers often assess the ability of a system to respond to a workload increase in terms of vertical scalability and horizontal scalability.

*Vertical scalability*, or scaling up, involves handling volume increases by running an application on larger, more powerful infrastructure resources. This scalability strategy was commonly used when monolithic applications were running on large servers such as mainframes or big Unix servers such as the Sun E10K. Changes to the application software or the database may be required when workloads increase in order to utilize increased server capacity (e.g., to take advantage of an increase in server memory).

Scalability is handled by the infrastructure, providing that a larger server is available, is affordable, and can be provisioned quickly enough to handle workload increases and that the application can take advantage of the infrastructure. This may be an expensive way to handle scaling, and it has limitations. However, vertical scaling is the only solution option for some problems such as in-memory data structures (e.g., graphs). It can be cost effective if the workload does not change quickly. The challenge is matching processing, memory, storage, and input/output (I/O) capacity to avoid bottlenecks. It is not necessarily a bad strategy.

*Horizontal scalability*, or scaling out, refers to scaling an application by distributing it on multiple virtual servers. This technique is often used as an

alternative to a vertical scalability approach, although it is also used to reduce latency and improve fault tolerance, among other goals. Several approaches may be used or even combined to achieve this. These classical approaches are still valid options, but containers and cloud-based databases provide new alternatives with additional flexibility:

- The simplest option involves segregating incoming traffic by some sort of partitioning, perhaps a business transaction identifier hash; by a characteristic of the workload (e.g., the first character of the security identifier); or by user group. This is a similar approach to the one used for sharding databases.<sup>9</sup> Using this option, a dedicated set of infrastructure resources, such as containers, handles a specific group of users. Because TFX will be deployed on a commercial cloud platform and is targeted to handle multiple banks using initially a multitenancy approach with full replication (see multitenancy considerations in the case study description in [Appendix A](#)), users could be segregated by bank (see [Figure 5.2](#)). Data would be distributed across separate tenant environments, as each bank would maintain its own set of TFX services and databases.

<sup>9</sup> For more information about database sharding, see Wikipedia, “Shard (Database Architecture).” [https://en.wikipedia.org/wiki/Shard\\_\(database\\_architecture\)](https://en.wikipedia.org/wiki/Shard_(database_architecture))



**Figure 5.2** *Distributing users for scalability*

- A second, more complex approach involves cloning the compute servers and replicating the databases. Incoming traffic is distributed

across the servers by a load balancer. Still, there are data challenges associated with this approach. All the data updates are usually made to one of the databases and then cascaded to all the others using a data replication mechanism. This process may cause update delays and temporary discrepancies between the databases (see the detailed discussion of the eventual consistency concept in [Chapter 3](#)). If the volume or frequency of database writes is high, it also causes the write database to become a bottleneck. For TFX, this option could be used for specific components, such as the Counterparty Manager (see [Figure 5.3](#)).



**Figure 5.3** Distributing TFX component clones for scalability

- A third, even more complex approach to horizontal scalability involves splitting the functionality of the application into services and distributing services and their associated data on separate infrastructure resource sets, such as containers (see [Figure 5.4](#)). This works well for TFX, as the design<sup>10</sup> is based on a set of services organized around business domains, following the Domain-Driven Design<sup>11</sup> approach, and it is deployed in cloud-based containers. Using this approach, data replication would be minimal because TFX data is associated to services and organized around business domains.

<sup>10</sup> See [Appendix A](#) for an outline of the TFX architectural design.

<sup>11</sup> The Domain-Driven Design approach drives the design of a software system from an evolving model of key business concepts. For more information, see <https://dddcommunity.org>. Also see Vaughn Vernon, *Implementing Domain-Driven Design* (Addison-Wesley, 2013).



**Figure 5.4** Distributing TFX services for scalability

## The Effect of Cloud Computing

Commercial cloud platforms provide a number of important capabilities, such as the ability to pay for resources that are being used and to rapidly scale when required. This is especially true when containers are being used, because infrastructure resources such as virtual machines may take significant time to start. The result may be that the system experiences issues while processing workloads for several minutes if capacity is reduced and then must be rapidly increased again. This is often a cost tradeoff, and it is one of the reasons containers have become so popular. They are relatively inexpensive in terms of runtime resources and can be started relatively quickly.

Cloud computing offers the promise of allowing an application to handle unexpected workloads at an affordable cost without any noticeable disruption in service to the application's customers, and as such, cloud computing is very important to scaling. However, designing for the cloud means more than packaging software into virtual machines or containers. For example, there are issues such as use of static IP addresses that require rework just to get the system working.

Although vertical scalability can be leveraged to some extent, horizontal scalability (called *elastic scalability* in cloud environments) is the preferred approach with cloud computing. Still, there are at least two concerns that need to be addressed with this approach. First, in a pay-per-use context,

unused resources should be released as workloads decrease, but not too quickly or in response to brief, temporary reductions. As we saw earlier in this section, using containers is preferable to using virtual machines in order to achieve this goal. Second, instantiating and releasing resources should preferably be automated to keep the cost of scalability as low as possible. Horizontal scalability approaches can carry a hidden cost in terms of human resources required to operate all the virtual servers required to handle the workload unless the operation of that infrastructure is fully automated.

Leveraging containers to deploy a suitably designed software system on cloud infrastructure has significant benefits over using virtual machines, including better performance, less memory usage, faster startup time, and lower cost. Design decisions such as packaging the TFX services and Web UI as containers, running in Kubernetes, ensure that TFX is designed to run natively on cloud infrastructures. In addition, the team has structured TFX as a set of independent runtime services, communicating only through well-defined interfaces. This design enables them to leverage horizontal scalability approaches.

Horizontal scalability approaches rely on using a load balancer of some sort (see [Figures 5.2](#) and [5.3](#)), for example, an API gateway and/or a service mesh,<sup>12</sup> as in the TFX architecture. In a commercial cloud, the costs associated with the load balancer itself are driven by the number of new and active requests and the data volumes processed. For TFX, scalability costs could be managed by leveraging an elastic load balancer. This type of load balancer constantly adjusts the number of containers and data node instances according to workload. Using this tool, infrastructure costs are minimized when workloads are small, and additional resources (and associated costs) are automatically added to the infrastructure when workloads increase. In addition, the team plans to implement a governance process to review the configurations of each infrastructure element periodically to ensure that each element is optimally configured for the current workload.

<sup>12</sup> A service mesh is a middleware layer that facilitates how services interact with each other.

An additional concern is that commercial cloud environments also may have scalability limits.<sup>13</sup> Software architects and engineers need to be aware of those limitations and create strategies to deal with them, for example, by

being able to rapidly port their applications to a different cloud provider if necessary. The tradeoff to such an approach is that you end up leveraging fewer of the cloud-native capabilities of the cloud provider of choice.

<sup>13</sup> Manzoor Mohammed and Thomas Ballard, “3 Actions Now to Prepare for Post-crisis Cloud Capacity.” <https://www.linkedin.com/smart-links/AQGGn0xF8B3t-A/bd2d4332-f7e6-4100-9da0-9234fdf64288>

## Architecting for Scalability: Architecture Tactics

How can a team leverage the Continuous Architecture approach to ensure that a software system is scalable? In this section, we discuss how the team plans to achieve overall scalability for TFX. We first focus on database scalability, because databases are often the hardest component to scale in a software system. Next, we review some scalability tactics, including data distribution, replication and partitioning, caching for scalability, and using asynchronous communications. We present additional application architecture considerations, including a discussion of stateless and stateful services as well as of microservices and serverless scalability. We conclude by discussing why monitoring is critical for scalability, and we outline some tactics for dealing with failure.

### TFX Scalability Requirements

For TFX, the client has good volume statistics and projections for its L/C business for the current and next two years. The team can leverage those estimates to document scalability in terms of stimulus, response, and measurement. Here’s an example of the documentation that they are creating:

- *Scenario 1 Stimulus:* The volume of issuances of import L/Cs increases by 10 percent every 6 months after TFX is implemented.
- *Scenario 1 Response:* TFX is able to cope with this volume increase. Response time and availability measurements do not change significantly.
- *Scenario 1 Measurement:* The cost of operating TFX in the cloud does not increase by more than 10 percent for each volume increase. Average response time does not increase by more than 5 percent

overall. Availability does not decrease by more than 2 percent. Refactoring the TFX architecture is not required.

- *Scenario 2 Stimulus:* The number of payment requests increases by 5 percent every 6 months after TFX is implemented, and the volume of issuances of export L/Cs remains roughly constant.
- *Scenario 2 Response:* TFX is able to cope with this increase in payment requests. Response time and availability measurements do not change significantly.
- *Scenario 2 Measurement:* The cost of operating TFX in the cloud does not increase by more than 5 percent for each volume increase. Average response time does not increase by more than 2 percent overall. Availability does not decrease by more than 3 percent. Refactoring the TFX architecture is not required.

Based on this documentation, the team feels that they have a good handle on the scalability requirements for TFX. Unfortunately, there is a significant risk associated with meeting these requirements. The plans to offer TFX to additional banks as a white label product are a major unknown at this time. Some of those banks have larger L/C volumes than those we have predicted, and the TFX marketing plans have not been finalized yet. TFX volumes and data may triple or even quadruple over the next few years, and the ability of TFX to handle this workload may become critical. In addition, TFX operating costs would likely impact its success. Each bank's portion of the fixed TFX costs may decrease as the number of banks using TFX increases, but a significant increase in variable costs due to a poor scalability design would more than offset this decrease. Should this situation happen, would banks quickly walk away from our platform? The team also wants to avoid a situation where scalability becomes a matter of urgency, or even survival, for TFX.

Given the uncertainty of marketing plans for TFX, the team decides to apply principle 3, *Delay design decisions until they are absolutely necessary*, and to architect the system for the known scalability requirements that they have already documented. Because the TFX architecture leverages principle 4, *Architect for change—leverage the “power of small,”* the team feels that enhancing TFX to handle additional banks if necessary would not be a major effort. For example, they are fairly

confident that they could leverage one of the three horizontal scalability approaches (or a combination of those approaches) to achieve this. They feel that it is better to avoid overarchitecting TFX for scalability requirements that may never materialize. They are also considering leveraging a standard library to handle interservice interfaces in order to be able to quickly address potential bottlenecks should workloads significantly increase beyond expectations (see “[Using Asynchronous Communications for Scalability](#)” later in this chapter).

The data tier of the architecture could also be another scalability pitfall. Distributing application logic across multiple containers is easier than distributing data across multiple database nodes. Replicating data across multiple databases can especially be challenging (see “[Data Distribution, Replication, and Partitioning](#)” later in this chapter). Fortunately, as described in [Appendix A](#), the TFX design is based on a set of services organized around business domains, following the Domain-Driven Design approach. This results in TFX data being associated to services and therefore organized around business domains. This design enables the data to be well partitioned across a series of loosely coupled databases and keeps replication needs to a minimum. The team feels that this approach should mitigate the risk of the data tier becoming a scalability bottleneck.

Identifying bottlenecks in TFX could be another challenge in solving scalability issues. It is expected that TFX users will occasionally experience response time degradation as workload increases. Based on the team’s recent experience with TFX, services that perform poorly at higher workloads are a bigger issue than those that fail outright, as they create a backlog of unprocessed requests that can eventually bring TFX to a halt. They need an approach to identify service slowdowns and failures and to remedy them rapidly.

As the size of the workload grows, rare conditions become more common, as the system is more likely to experience performance and scalability edge cases as it approaches its limits. For example, hardcoding limits, such as hardcoding the maximum number of documents related to an L/C, can cause serious problems, creating scalability issues that manifest without warning. One such example, although a little dated, is what happened to U.S. airline Comair during the 2004 holiday season. The crew-scheduling system crashed, which, along with severe weather, caused large numbers of

flight cancellations and necessitating crew rescheduling.<sup>14</sup> The system had a limit of 32,000 crew-assignment inquiries a month, by design. We can only assume that the system designers and their business stakeholders felt that that limit was more than adequate when the system was first implemented 10 years earlier, and no one thought about adjusting it as the system was handling larger workloads.<sup>15</sup>

<sup>14</sup> Julie Schmit, “Comair to Replace Old System That Failed,” *USA Today* (December 28, 2004). [https://usatoday30.usatoday.com/money/biztravel/2004-12-28-comair-usat\\_x.htm](https://usatoday30.usatoday.com/money/biztravel/2004-12-28-comair-usat_x.htm)

<sup>15</sup> Stephanie Overby, “Comair’s Christmas Disaster: Bound to Fail,” *CIO Magazine* (May 1, 2005). <https://www.cio.com/article/2438920/comair-s-christmas-disaster--bound-to-fail.html>

Finally, we also need to consider the scalability of operational activities and processes because operations concerns are tied to actionable architecture concerns. Infrastructure configuration management could become an issue as infrastructure resources are added to cope with increased workloads. As the number of containers and database nodes grows significantly, monitoring and managing all these infrastructure resources becomes harder. For example, the software would need to be either upgraded fairly rapidly to avoid having multiple versions in production concurrently or architected to allow multiple versions operating side by side. Security patches would especially need to be quickly applied to all infrastructure elements. Those tasks need to be automated for a scalable system.

## Database Scalability

As noted in the previous section, the TFX team has had some discussions about the ability of the TFX Contract Management database to cope with large workloads, especially if the TFX adoption rate would cause those workloads to exceed estimates. Theoretically, the design should enable the TFX platform to scale, because its data is well partitioned across a series of loosely coupled databases. In addition, data sharing and replication needs have been kept to a minimum. This approach should mitigate the risk of the data tier becoming a scalability bottleneck. However, the team decides to assess the upper scalability limit of the architecture by running an early scalability test.

The TFX team leverages principle 5, *Architect for build, test, deploy, and operate*, and decides to start testing for scalability as early in the development cycle as possible. The team designs and builds a simplified, bare-bones prototype of the TFX platform that implements only a few key transactions, such as issuing a simple L/C and paying it. Using this prototype, the team runs several stress tests<sup>16</sup> using simulations of the expected TFX transaction mix. The workload used for these stress tests is based on volume projections for this year and the next 2 years, plus a 100 percent safety margin. Unfortunately, as the test workload increases, some of the databases associated with several TFX services, such as the Contract Manager and the Counterparty Manager, become bottlenecks and affect the performance of the overall platform. Services accessing and updating those components experience noticeable slowdowns, which have adverse repercussions on the performance and availability of the TFX platform.

<sup>16</sup> See [Chapter 6, “Performance as an Architectural Concern.”](#)

The team verifies that database queries, especially those related to reporting, are a large part of the expected TFX workload and create scalability concerns. They take some initial steps, such as optimizing queries and reconfiguring the services for more capacity at increased cost, in order to facilitate vertical scaling. Those steps immediately help with the stress testing, but do not suffice as the test workloads are increased further. In addition, it proves to be hard to optimize the TFX database design for both the updates and queries that it needs to support. After some analysis, they conclude that attempting to support both workloads with a single database for each service could result in a compromised design that may not perform sufficiently well for either mode. This may also create performance and even availability issues as the TFX workload increases.

Given this challenge, one option the team can select is to use a separate database to process TFX queries related to reporting requirements (described as path C in [Chapter 3](#)). This database, called the analytics database (see [Chapter 3](#) for more details on that database), ingests TFX transactional data and stores it in a format optimized to handle the TFX reporting requirements (see [Figure 5.5](#)). As the TFX transactional database is updated, the updates are replicated to the analytics database using the TFX database management system (DBMS) replication mechanism

process. Another approach considered by the team would be to use an event bus<sup>17</sup> to replicate the updates. However, after analyzing both options (see the architectural decision log, [Table A.2](#), in [Appendix A](#)) and evaluating their tradeoffs, the team feels that using an event bus may increase latencies by milliseconds up to seconds at high volumes due to the data serialization, data transmission, data deserialization, and write processing. Using the TFX DBMS replication mechanism would reduce the propagation latency because, in most cases, it would be shipping database logs rather than processing events. This is very important for consistency and scalability.

<sup>17</sup> An event bus is a type of middleware that implements a publish/subscribe pattern of communications between services. It receives events from a source and routes them to another service interested in consuming those events.



**Figure 5.5** Scaling a TFX service for high workloads

This new approach results in allowing TFX to handle higher workloads without issues, and additional stress tests are successful. However, the TFX data eventually may need to be distributed to cope with volumes beyond what the team expects for the current and next 2 years if the company is successful in getting other banks to adopt TFX. The team may be able to handle additional volumes by cloning the TFX analytics database and running each instance on a separate node without any major changes to the architecture.

Since the initial deployment of TFX will be a multitenancy installation with full replication (see multitenancy considerations in [Appendix A](#)), the team believes that the database tactics depicted in [Figure 5.5](#) are sufficient to

handle the expected workloads. However, as mentioned in the case study architectural description in [Appendix A](#), a full replication approach becomes harder to administer as the number of banks using TFX increases. Each bank installation needs to be monitored and managed independently, and the software for each installation needs to be upgraded reasonably quickly to avoid having many versions in production simultaneously. The team may need to consider switching to other multitenancy approaches, such as data store replication or even full sharing, at some point in the future. This potentially means distributing, replicating, or partitioning TFX data.

## Data Distribution, Replication, and Partitioning

Using the TFX prototype developed for stress testing, let us follow the team in a scenario where they explore a few data architecture options to better understand their tradeoffs in terms of benefits versus costs, meaning effort to implement and to maintain.

The team first looks at implementing the approach of cloning the compute servers and sharing the databases described under “[Types and Misunderstandings of Scalability](#)” earlier in this chapter. They clone a few TFX services that may experience scalability issues, such as Counterparty Manager, and run each clone on a separate set of containers (see [Figure 5.3](#)). As depicted in [Figure 5.3](#), all of the database updates are done on one of the instances (database master), and those updates are replicated to the other databases using the DBMS replication process. Stress testing shows that this approach yields some scalability benefits over their current architecture, but it does not solve the problem of managing multiple installations of the services that are cloned.

As an alternative option, they decide to look into partitioning the TFX data for specific services, such as the Counterparty Manager. This involves splitting the database rows using some criteria, for example, user group. In this case, it may make sense to partition the rows by bank. As their next step, they implement table partitioning for the Counterparty Manager and the ContractManager databases, using the TFX prototype. They then run a stress test using a scenario with workloads that they would expect five large banks and five small ones to generate. This test is successful, and they believe that they have a viable architectural approach to implement

multitenancy without using full replication should the need arise. However, using table partitioning increases architectural complexity. Database design changes become more complicated when this capability is used. We should use database partitioning only when all other options for scaling our database have been exhausted.

For TFX, the team would probably opt for a managed cloud-based alternative should they need to implement data partitioning or even sharding. This approach would, of course, increase the cost of implementing the solution as well as its ongoing operating cost and possibly a large migration cost to a different database model. As we mentioned earlier in this chapter, architectural decisions for scalability have a significant impact on deployment costs. When selecting an architecture to increase the scalability of TFX, we need to make a tradeoff with other quality attributes, such as cost in this case.

As stated earlier and in [Appendix A](#), because the initial TFX deployment will be a full replication installation, the team applies principle 3, *Delay design decisions until they are absolutely necessary*, and decides not to implement table partitioning at this time (see architectural decision log, [Table A.2](#), in [Appendix A](#)). However, the team will need to reassess using table partitioning or even sharding when it becomes necessary to switch to other multitenancy approaches as the number of banks using TFX increases.

## Caching for Scalability

In addition to highlighting some database issues, let us assume that the team's earlier stress test uncovered some potential application related performance issues as well. As described under "[Database Scalability](#)," the team stress tested the TFX platform prototype with simulations of the expected TFX transaction mix based on volume projections for this year and the next 2 years plus a 100 percent safety margin. Specifically, they observed that a few TFX transactions, such as L/C payment, using early versions of the Counterparty Manager, the Contract Manager, and the Fees and Commissions Manager services, were experiencing some unexpected slowdowns. What should they do about this? As Abbott and Fisher point out, "The absolute best way to handle large traffic volumes and user

requests is to not have to handle it at all. . . . The key to achieving this is through the pervasive use of something called a cache.”<sup>18</sup>

<sup>18</sup> Abbott and Fisher, *The Art of Scalability*, 395.

Caching is a powerful technique for solving some performance and scalability issues. It can be thought of as a method of saving results of a query or calculation for later reuse. This technique has a few tradeoffs, including more complicated failure modes, as well as the need to implement a cache invalidation process to ensure that obsolete data is either updated or removed. There are many caching technologies and tools available, and covering all of them is beyond the scope of this book. Let us look at four common caching techniques and investigate how they may help with the TFX scalability challenge.

- *Database object cache*: This technique is used to fetch the results of a database query and store them in memory. For TFX, a database object cache could be implemented using a caching tool such as Redis or Memcached. In addition to providing read access to the data, caching utilities provide their clients with the ability to update data in cache. To isolate the TFX service from the database object caching tool, the team has the option of implementing a simple data access API. This API would first check the cache when data is requested and access the database (and update the cache accordingly) if the requested data isn’t already in cache. It also would ensure that the database is updated when data is updated in cache. The team runs several tests that show that the database object cache has a positive impact on scalability, as it eliminates the need for a process to access the TFX database and deserialize objects if they are already in cache. The TFX team could implement this technique for several TFX services. Because early scalability testing has shown that some TFX transactions, such as L/C payment, may experience some performance challenges, they consider implementing database object cache for the Counterparty Manager, the Contract Manager, and the Fees and Commissions Manager components.
- *Application object cache*: This technique stores the results of a service, which uses heavy computational resources, in cache for later retrieval by a client process. This technique is useful because, by caching

calculated results, it prevents application servers from having to recalculate the same data. However, the team does not yet have any use case that could benefit from this technique, so they decide to defer implementing application object cache for now.

- *Proxy cache*: This technique is used to cache retrieved Web pages on a proxy server<sup>19</sup> so that they can be quickly accessed next time they are requested, whether by the same user or by a different user. Implementing it involves some changes to the configuration of the proxy servers and does not require any changes to the TFX software system code. It may provide some valuable scalability benefits at a modest cost, but the team does not know yet of any specific issues that this technique would address, so they decide to defer its implementation following principle 3, *Delay design decisions until they are absolutely necessary*.
- *Precompute cache*: This technique stores the results of complex queries on a database node for later retrieval by a client process. For example, a complex calculation using a daily currency exchange rate could benefit from this technique. In addition, query results can be cached in the database object cache if necessary. Precompute cache is different from standard database caching provided by DBMS engines. Materialized views (see “[Modern Database Performance Tactics](#)” in [Chapter 6](#)), which are supported by all traditional SQL databases, are a type of precompute cache. Precomputing via database triggers is another example of precompute cache. The team does not know yet of any TFX use case that could benefit from this technique, so they decide to defer implementing it.

<sup>19</sup> Proxy servers act as intermediaries for requests from enterprise clients that need to access servers external to the enterprise. For more details, see Wikipedia, “Proxy Server,” [https://en.wikipedia.org/wiki/Proxy\\_server](https://en.wikipedia.org/wiki/Proxy_server)

Two additional forms of caching may already be present in the TFX environment. The first one is a static (browser) cache, used when a browser requests a resource, such as a document. The Web server requests the resource from the TFX system and provides it to the browser. The browser uses its local copy for successive requests for the same resource rather than retrieving it from the Web server. The second one is a content delivery

network (CDN)<sup>20</sup> that could provide the JavaScript code and that can be used for static resources in case of a Web tier scalability problem. The benefits of a CDN are that it would cache static content close to the TFX customers regardless of their geographical location. However, because a CDN is designed for caching relatively static content, it is unclear whether its benefits would outweigh its costs for TFX.

<sup>20</sup> AWS CloudFront is an example of a CDN.

## Using Asynchronous Communications for Scalability

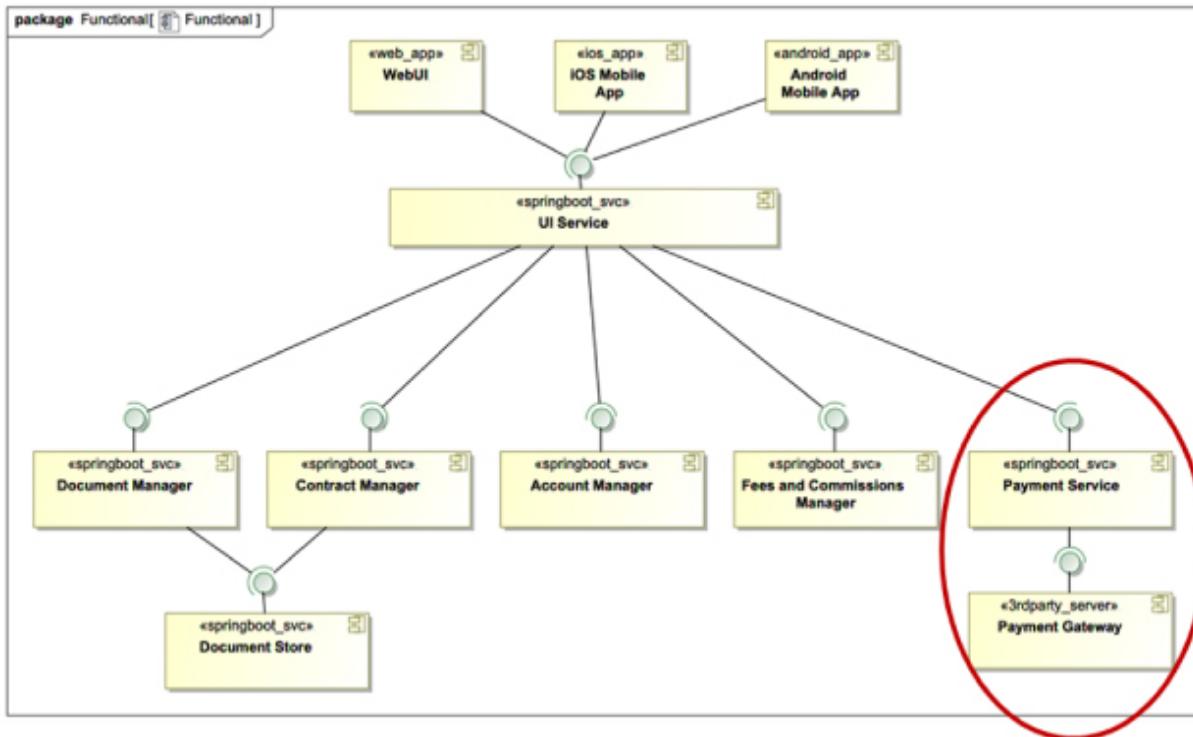
Initially, the TFX architecture assumes that most interservice interactions are synchronous, using the REST<sup>21</sup> architecture style (see the architectural description of the case study in [Appendix A](#) for details). What do we mean by *synchronous communications*? If a service request is synchronous, it means that code execution will block (or wait) for the request to return a response before continuing. Asynchronous interservice interactions do not block (or wait) for the request to return from the service in order to continue.<sup>22</sup> However, retrieving the results of the service execution involves extra design and implementation work. In a nutshell, synchronous interactions are simpler and less costly to design and implement than asynchronous ones, which is the rationale behind using them in the initial TFX architecture.

<sup>21</sup> REST is a lightweight, HTTP-based architecture style used for designing and implementing Web services. For more details, see Wikipedia, “Representational State Transfer,” [https://en.wikipedia.org/wiki/Representational\\_state\\_transfer](https://en.wikipedia.org/wiki/Representational_state_transfer).

<sup>22</sup> For more details on this topic, see Abbott and Fisher, *The Art of Scalability*.

Unfortunately, the team just introduced a new scalability risk with this approach, and this risk needs to be addressed. Scalability is a key quality attribute for TFX, and we cannot ignore principle 2, *Focus on quality attributes, not on functional requirements*. Since synchronous interactions stop the execution of the requesting process until the software component being invoked completes its execution, they may create bottlenecks as the TFX workload increases. In a synchronous interaction model, there is only one request in flight at a time. In an asynchronous model, there may be more than one request in flight. An asynchronous approach improves

scalability if the request-to-response time is long, but there is adequate request processing throughput (requests per second). It can't help if there is inadequate request processing throughput, although it may help smooth over spikes in request volume. The team may need to switch from synchronous interactions to asynchronous ones for TFX components that are likely to experience resource contentions as workloads increase. Specifically, they suspect that the requests to the Payment Service may need to be switched to an asynchronous mode, as this service interfaces with a Payment Gateway that they do not control, as it is third-party software and may experience delays with high workloads (see [Figure 5.6](#)).



**Figure 5.6** TFX component diagram

To avoid a major rework of the TFX application, should this switch become necessary, the team decides to use a standard library for all interservice interactions. Initially, this library implements synchronous interactions using the REST architectural style. However, it can be switched to an asynchronous mode for selected interservice communications should the need arise. To mitigate concurrency issues that may be introduced by this switch, the library will use a message bus<sup>23</sup> for asynchronous requests. In addition, the library implements a standard interface monitoring approach

in order to make sure that interservice communication bottlenecks are quickly identified (see [Figure 5.7](#)). An important characteristic of the TFX asynchronous service integration architecture is that, unlike an enterprise service bus (ESB) that is usually used to implement a service-oriented architecture (SOA), the message bus used in TFX has only one function, which is to deliver messages from publishing services such as the Payment Service to subscribing services such as the Payment Gateway. Any message processing logic is implemented in the services that use the message bus, following what is sometimes called a smart endpoints and dumb pipes<sup>24</sup> design pattern. This approach decreases the dependency of services on the message bus and enables the use of a simpler communication bus between services.

<sup>23</sup> For example, AWS EventBridge.

<sup>24</sup> Martin Fowler, “Microservices.”  
<https://martinfowler.com/articles/microservices.html#SmartEndpointsAndDumbPipes>



**Figure 5.7** *TFX asynchronous service integration architecture*

Another example of where asynchronous communications can be applied was initially introduced in [Chapter 3](#). Here is a brief summary of that example: the TFX team decides to prototype a solution based on using sensors affixed to shipping containers in order to better track the shipment of goods. They partner with a company that deals with the complexity of physically installing and tracking sensors for each shipment. These sensors generate data that need to be integrated with TFX. The scalability

requirements of the TFX component responsible for this integration are unknown at this point. However, given the amount of data that is likely to be generated by the sensors, the TFX team decides to use an asynchronous communication architecture for this integration. In addition, this approach would enable the TFX platform to integrate with several other sensor providers should the organization decide to partner with them as well.

## Additional Application Architecture Considerations

Presenting an exhaustive list of application architecture and software engineering considerations to develop scalable software is beyond the scope of this book. Some of the materials mentioned under “[Further Reading](#)” provide additional information on this topic. However, the following are a few additional guidelines for ensuring scalability.

### *Stateless and Stateful Services*

Software engineers may engage in animated discussions about whether a service is stateless or stateful, especially during architecture and design reviews. During those discussions, stateless services are usually considered good for scalability, whereas stateful services are not. But what do we mean by *stateless* and *stateful*? Simply put, a stateful service is a service that needs additional data (usually the data from a previous request) besides the data provided with the current request in order to successfully execute that request. That additional data is referred to as the *state* of the service. User session data, including user information and permissions, is an example of state. There are three places where state can be maintained: in the client (e.g., cookies), in the service instance, or outside the instance. Only the second one is stateful. A stateless service does not need any additional data beyond what is provided with the request,<sup>25</sup> usually referred to as the *request payload*. So why do stateful services create scalability challenges?

<sup>25</sup> For example, HTTP is considered stateless because it doesn’t need any data from the previous request to successfully execute the next one.

Stateful services need to store their state in the memory of the server they are running on. This isn’t a major issue if vertical scalability is used to handle higher workloads, as the stateful service would execute successive requests on the same server, although memory usage on that server may

become a concern as workloads increase. As we saw earlier in this chapter, horizontal scalability is the preferred way of scaling for a cloud-based application such as TFX. Using this approach, a service instance may be assigned to a different server to process a new request, as determined by the load balancer. This would cause a stateful service instance processing the request not to be able to access the state and therefore not to be able to execute correctly. One possible remedy would be to ensure that requests to stateful services retain the same service instance between requests regardless of the server load. This could be acceptable if an application includes a few seldom-used stateful services, but it would create issues otherwise as workloads increase. Another potential remedy would be to use a variant of the first approach for implementing horizontal scalability, covered earlier in this chapter. The TFX team's approach would be to assign a user to a resource instance for the duration of a session based on some criteria, such as the user identification, thereby ensuring that all stateful services invoked during that session execute on the same resource instance. Unfortunately, this could lead to overutilization of some resource instances and underutilization of others, because traffic assignment to resource instances would not be based their load.

It would clearly be better to design and implement stateless services for TFX rather than stateful ones. However, the team faces two challenges with mandating that all TFX services be stateless. The first one is that their software engineers are used to working with stateful services due to their training, design knowledge, experience, and the tools that they are using. Familiarity with stateful service design makes creating those services easier and simpler than creating stateless ones, and engineers need time to adapt their practices accordingly.

The second challenge deals with accessing user session data from a stateless service. This requires some planning, but it's an easier challenge to solve than the first one. After conducting a design session, the team decides to keep the size of the user session data object to a minimum and to cache<sup>26</sup> it (see “[Caching for Scalability](#)” earlier in this chapter). A small user session data item can be quickly transmitted between servers in order to facilitate distributed access, and caching that data ensures that stateless processes can quickly access it using a reference key included in each request payload.

<sup>26</sup> Using a caching tool such as Redis or Memcached.

Using this pattern, the team should be able to make the vast majority of TFX services stateless. Stateful exceptions should be reviewed and approved by the team.

## ***Microservices and Serverless Scalability***

In chapter 2 of the original *Continuous Architecture* book, we wrote the following about using microservices in order to implement principle 4, *Architect for change—leverage the “power of small”*:

Using this approach, many services are designed as small, simple units of code with as few responsibilities as possible (a single responsibility would be optimal), but leveraged together can become extremely powerful. The “Microservice” approach can be thought of as an evolution of Service Oriented Architectures (SOAs). . . . Using this design philosophy, the system needs to be architected so that each of its capabilities must be consumable independently and on demand. The concept behind this design approach is that applications should be built from components that do a few things well, are easily understandable—and are easily replaceable should requirements change. Those components should be easy to understand, and small enough to be thrown away and replaced if necessary. . . . Microservices can be leveraged for designing those parts of the system that are most likely to change—and therefore making the entire application more resilient to change. Microservices are a critical tool in the Continuous Architecture toolbox, as they enable loose coupling of services as well as replaceability—and therefore quick and reliable delivery of new functionality.<sup>27</sup>

<sup>27</sup> Erder and Pureur, *Continuous Architecture*, 31.

We believe that what we wrote is still applicable to microservices today, especially the part on thinking of this approach as a refinement of the SOA approach to create loosely coupled components with as few responsibilities as possible. The challenge with the SOA approach may have been that it was hierarchical and defined in a rigid manner with a protocol (WSDL)<sup>28</sup> that was inflexible. Microservices evolved from this approach and use

simpler integration constructs such as RESTful APIs. The term *microservice* may be misleading, as it implies that those components should be very small. Among the microservices characteristics, size has turned out to be much less important than loose coupling, stateless design, and doing a few things well. Leveraging a Domain-Driven Design approach, including applying the bounded context pattern to determine the service boundaries, is a good way to organize microservices.

<sup>28</sup> Web Service Definition Language. For more details, see <https://www.w3.org/TR/wsdl.html>.

Microservices communicate with each other using a lightweight protocol such as HTTP for synchronous communications or a simple message or event bus that follows the smart endpoints and dumb pipes pattern for asynchronous ones (refer to “[Using Asynchronous Communications for Scalability](#)”). A microservices-based architecture can fairly easily be scaled using any of the three techniques described in our discussion of horizontal scalability earlier in this chapter.

What about serverless computing, which has emerged since we wrote the original *Continuous Architecture* and is also known as Function as a Service (FaaS)? FaaS can be thought of as a cloud-based model in which the cloud provider manages the computing environment that runs customer-written functions as well as manages the allocation of resources. Serverless functions (called lambdas [ $\lambda$ ] on Amazon’s AWS cloud) are attractive because of their ability to autoscale, both up and down, and their pay-per-use pricing model. Using serverless computing, a software engineer can ignore infrastructure concerns such as provisioning resources, maintaining the software, and operating software applications. Instead, software engineers can focus on developing application software. Serverless functions can be mixed with more traditional components, such as microservices, or the entire software system can even be composed of serverless functions.

Of course, serverless functions do not exist in isolation. Their use depends on using a number of components provided by the cloud vendor, collectively referred to as the *serverless architecture*. A full description of a serverless architecture is beyond the scope of this book;<sup>29</sup> however, it is important to point out that those architectures usually increase the dependency of the application on the cloud vendor because of the number

of vendor-specific components that they use. Applications that leverage a serverless architecture provided by a cloud vendor are usually not easily ported to another cloud.

<sup>29</sup> For more information, see Martin Fowler, “Serverless Architectures.” <https://martinfowler.com/articles/serverless.html>

From an architectural perspective, serverless functions should have as few responsibilities as possible (one is ideal) and should be loosely coupled as well as stateless. So, what is the benefit microservices beyond the fact that software engineers do not need to provision servers for serverless functions? It has to do with the way microservices interface with each other compared to serverless functions. Microservices mostly communicate using a request/reply model, although they can be designed to communicate asynchronously using a message bus or an event bus. Serverless functions are event based, although they can also be invoked using a request/reply model through an API gateway. A common mode of operation is to be triggered by a database event, such as a database update, or by an event published on an event bus. [Figure 5.8](#) illustrates various ways of invoking serverless functions. It depicts the following scenarios:

- Function Lambda 1 is invoked when a database update is made by Microservice 1.
- Function Lambda 2 is invoked when Microservice 2 publishes an application to an event bus.
- Finally, both Lambda 1 and Lambda 2 can also be directly invoked via an API gateway.

The team has the option to use a few serverless functions as part of the TFX system, mainly to provide notifications. For example, a serverless function could notify both the importer and the exporter that an L/C has been issued and that documents have been presented for payment.



**Figure 5.8** Combining microservices and serverless functions

Serverless functions such as lambdas are scalable for some workloads, but other workloads can be challenging. They tend to be dependent on other services, so the team needs to make sure they don't accidentally cause a scalability problem by calling a service that doesn't scale as well as lambdas do. In addition, the serverless event-driven model may create design challenges for architects who are not used to this model. If not careful, utilizing serverless functions can create a modern spaghetti architecture with complex and unmanageable dependencies between functions. On the other hand, a serverless architecture can react more rapidly to unexpected workload spikes, as there are no delays in provisioning additional infrastructure. [Table 5.1](#) summarizes the differences and tradeoffs between microservices and serverless architecture styles.

**Table 5.1** Comparison of Microservices and Serverless Architecture Styles

<b>Element</b>	<b>Microservices</b>	<b>Serverless</b>
Architecture style	Primarily service based architecture (can also support event based)	Primarily event-based architecture (can also support service based). Serverless functions can also act as scheduled jobs.
Components	Services	Functions
Technology maturity	Mature technology with tools and processes	Less mature and evolving rapidly. Tightly coupled to the provider, and the technology is changing in ways that affect performance and other qualities. You need to be ready to keep up with someone else's roadmap.
Scalability	Medium to high—depends on architecture and implementation	Great scalability for some workloads, but others can be challenging. Limited by the provider, as are the processor and memory profiles of the execution environment. Also can be limited by the scalability of other architecture components. Unpredictable cold-start latency can be an issue for some workloads.
Openness	Open technology	Not open. Need to buy into the providers' infrastructure not only for the serverless function execution but for all the other cloud services that you need for security, request routing, monitoring, persistence, etc.
Programming language support	Supports most modern languages	Fewer languages supported
Granularity	High to low	Higher than microservices
State management	Both stateful and stateless modes are available	Stateless by design, but the instance life cycle is not under your control, and there is no option for sticky sessions.
Deployment time	Architecture and implementation dependent	Serverless is faster if you don't have a microservices infrastructure already in place.
Cost model	Depends on implementation	Pay-as-you-go
Operational costs	Implementation dependent	Could be lower than for microservices
<b>Overall tradeoff</b>	<b>You design and operate your</b>	<b>You delegate all the architecture and operations</b>

<b>Overall tradeoff</b>	You design and operate your microservices infrastructure and develop and deploy services that are compatible with that infrastructure. However, microservices can get bound to the hosting cloud infrastructure as well.	You delegate <i>all</i> the architecture and operations decisions to the provider, and you buy into its <i>entire</i> platform, since serverless functions need a lot of surrounding services to work. If you fit the provider's target use case, things will be smooth and easy. If you don't fit the use case (or more significantly, if you evolve so that you no longer fit the use case), then it becomes hard, and there is no easy migration path from serverless to something else.
-------------------------	--	---

## Achieving Scalability for TFX

The TFX team is planning to achieve scalability based on current requirements by applying a number of architecture tactics, which are discussed earlier in this chapter. In addition, monitoring the system for scalability and dealing with failure are two key aspects of achieving scalability.

We briefly recapitulate some of the tactics they plan to use for this purpose (see [Table 5.2](#)). Those tactics are discussed in detail in this chapter.

**Table 5.2** Sample TFX Scalability Tactics

Tactic	TFX Implementation
Database scalability tactics	Focus on database scalability because databases are often the hardest component to scale in a software system. Use data distribution and replication when necessary. Delay using complex approaches such as data partitioning and sharding until there are no other options.
Caching	Leverage caching approaches and tools, including database object caching and proxy caching.
Stateless services	Use stateless services and microservices when possible. Stateful services should be used when no other option exists.
Microservices and serverless	Serverless functions should be used for scalability when it makes sense to do so.
Monitoring TFX scalability	Design and implement a monitoring architecture, which is critical for scalability.
Dealing with failure	Use specific tactics for dealing with failure, such as circuit breakers, governors, timeouts, and bulkheads.

The team believes that this approach will enable the TFX system to scale adequately and meet both current requirements and future expansion plans.

## ***Measuring TFX Scalability***

Monitoring is a fundamental aspect of the TFX system that is needed to ensure that it is scalable. Monitoring is discussed in detail in the “[Operational Visibility](#)” section of [Chapter 7](#), so we include only a brief summary in this section.

As we saw earlier in this chapter, it is hard to predict exactly how large a transaction volume TFX will have to handle. In addition, determining a realistic transaction mix for that workload may be an even bigger challenge. As a result, it is hard to load and stress test a complex software system like TFX in a realistic manner. This does not mean that load and stress testing, described in [Chapter 6](#), are not useful or should not be used, especially if an effort has been made to document scalability requirements as accurately as possible based on business estimates. However, testing should not be the only way to ensure that TFX will be able to handle high workloads.

[Figure 5.9](#) provides a high-level overview of the TFX scalability monitoring architecture.



**Figure 5.9** *TFX scalability monitoring architecture overview*

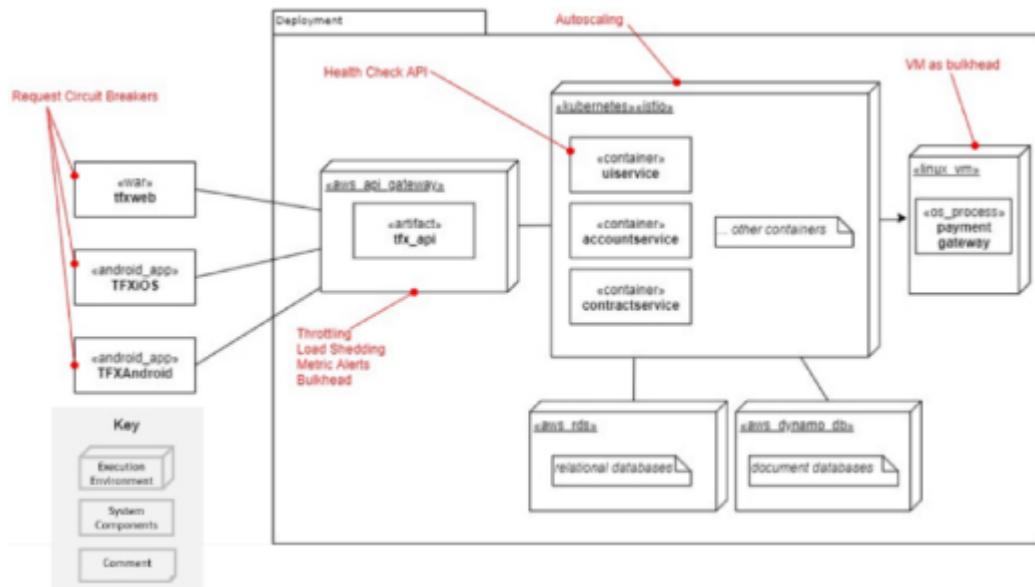
Effective logging, metrics, and the associated automation are critical TFX components of the monitoring architecture. They enable the team to make data-driven scalability decisions when required and to cope with unexpected spikes in the TFX workload. The team needs to know precisely which TFX components start experiencing performance issues at higher workloads and to be able to take remedial action before the performance of the whole TFX system becomes unacceptable. The effort and cost of designing and implementing an effective monitoring architecture is part of the overall cost of making TFX scalable.

### ***Dealing with Scalability Failure***

Why do we need to deal with failure caused by scalability issues? System component failures are inevitable, and the TFX system must be architected to be resilient. Large Internet-based companies such as Google and Netflix have published information on how they deal with that challenge.<sup>30</sup> But why is the TFX monitoring architecture not sufficient for addressing this issue? Unfortunately, even the best monitoring framework does not prevent failure, although it would send alerts when a component fails and starts a chain reaction that could bring down the whole platform. Leveraging principle 5, *Architect for build, test, deploy, and operate*, the team applies additional tactics to deal with failure. Those tactics are described in detail in

the “Dealing with Incidents” section of [Chapter 7](#), so we include only a brief overview in this section. Examples of these tactics include implementing circuit breakers, throttling, load shedding, autoscaling, and bulkheads as well as ensuring that system components fail fast, if necessary, and performing regular health checks on system components (see [Figure 5.10](#)).

<sup>30</sup> Please see, for example, “What Is Site Reliability Engineering (SRE)?” <https://sre.google>, and Yury Izrailevsky and Ariel Tseitlin, “The Netflix Simian Army” [blog post]. <https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116>



**Figure 5.10** Overview of tactics for dealing with scalability failure

It is important to point out that all the TFX components that deal with failure are integrated into the TFX monitoring architecture. They are an essential source of information for the monitoring dashboard and for the automation components of that architecture.

## Summary

This chapter discussed scalability in the context of Continuous Architecture. Scalability has not been traditionally at the top of the list of quality attributes list for an architecture. However, this has changed during the last few years, perhaps because large Internet-based companies such as

Google, Amazon, and Netflix have been focusing on scalability. Software architects and engineers realize that treating scalability as an afterthought may have serious consequences should an application workload suddenly exceed expectations. When this happens, architects and software engineers are forced to make decisions rapidly, often with insufficient data.

Dealing with other high-priority quality attribute requirements, such as security, performance, and availability, is usually hard enough.

Unfortunately, unforeseen workload spikes sometimes propel scalability to the top of the priority list in order to keep the system operational. Like any other quality attribute, scalability is achieved by making tradeoffs and compromises, and scalability requirements need to be documented as accurately as possible. In this chapter, we covered the following topics:

- Scalability as an architectural concern. We reviewed what has changed over the last few years. We discussed the assumption of scalability, the forces affecting scalability, the types of scalability (including horizontal scalability and vertical scalability), some misunderstandings of scalability, and the effect of cloud computing on scalability.
- Architecting for scalability. Using the TFX case study, we presented some common scalability pitfalls. We focused on database scalability because databases are often the hardest component to scale in a software system. We reviewed some scalability tactics, including data distribution, replication and partitioning, caching for scalability, and using asynchronous communications for scalability. We presented additional application architecture considerations, including a discussion of stateless and stateful services as well as of microservices and serverless scalability. We finally discussed why monitoring is critical for scalability and gave a brief overview of some tactics for dealing with failure.

Scalability is distinct from performance. Scalability is the property of a system to handle an increased (or decreased) workload by increasing (or decreasing) the cost of the system. Performance is “about time and the software system’s ability to meet its timing requirements.”<sup>31</sup> In addition, a complete perspective for scalability requires us to look at business, social, and process considerations as well as its architectural and technical aspects.

<sup>31</sup> Bass, Clements, and Kazman, *Software Architecture in Practice*, 131.

A number of proven architectural tactics can be used to ensure that modern systems are scalable, and we presented some of the key ones in this chapter. However, it is also important to remember that performance, scalability, resilience, usability, and cost are tightly related and that scalability cannot be optimized independently of the other quality attributes. Scenarios are an excellent way to document scalability requirements, and they help the team create software systems that meet their objectives.

It is important to remember that for cloud-based systems, scalability (like performance) isn't the problem of the cloud provider. Software systems need to be architected to be scalable, and porting a system with scalability issues to a commercial cloud will probably not solve those issues.

Finally, to provide acceptable scalability, the architecture of a modern software system needs to include mechanisms to monitor its scalability and deal with failure as quickly as possible. This enables software practitioners to better understand the scalability profile of each component and to quickly identify the source of potential scalability issues.

[Chapter 6](#) discusses performance in the Continuous Architecture context.

## Further Reading

This section includes a list of books and websites that we have found helpful to further our knowledge in the scalability space.

- Martin L. Abbott and Michael T. Fisher's *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise* (Addison-Wesley, 2015) is a classic textbook that extensively covers most aspects of scalability, including organizations and people, processes, architecting for scalability, emerging technologies and challenges, as well as measuring availability, capacity, load, and performance.
- John Allspaw and Jesse Robbins's *Web Operations: Keeping the Data on Time* (O'Reilly Media, 2010) has a lot of good information, based on practical experience, on how to operate web sites and deal with a number of challenges, including scalability, from an operational viewpoint.

- In *Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems* (O'Reilly Media, 2017), Martin Kleppmann discusses scalability, reliability, and performance for data-oriented applications. It is a classic text of its type.
- Michael T. Nygard's *Release It!: Design and Deploy Production-Ready Software* (Pragmatic Bookshelf, 2018) is primarily about resilience but contains a lot of information that is relevant to scalability as well.
- Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy's *Site Reliability Engineering: How Google Runs Production Systems* (O'Reilly Media, 2016) is a useful book on recognizing, handling, and resolving production incidents, including scalability issues.
- Ana Oprea, Betsy Beyer, Paul Blankinship, Heather Adkins, Piotr Lewandowski, and Adam Stubblefield's *Building Secure and Reliable Systems* (O'Reilly Media, 2020) is a newer book from a Google team. This book focuses on system design and build for creating secure, reliable, and scalable systems.
- High Scalability (<http://highscalability.com>) is a great resource for information on scalability of websites. This site directs you to many tutorials, books, papers, and presentations and is a great place to start. All of the advice is practical and is based on the experiences of large Internet companies.

# Chapter 6. Performance as an Architectural Concern

*You can use an eraser on the drafting table or a sledge hammer on the construction site.*

—Frank Lloyd Wright

Unlike scalability, performance has traditionally been at the top of the list of quality attributes for an architecture. There is no need to convince software practitioners to take performance into serious consideration when they design an architecture. They know that poor performance may have a serious impact on the usability of the software system.

This chapter considers performance in the context of Continuous Architecture, discussing performance as an architectural concern as well as how to architect for performance. We do not discuss detailed performance engineering concerns unless they are relevant to architecture. Because performance is such a large field,<sup>1</sup> we can only touch on some aspects that are relevant to the Continuous Architecture context and the Trade Finance eXchange (TFX) case study. Using the TFX case study, this chapter provides practical examples and actionable advice.

<sup>1</sup> For more information on performance and especially on performance engineering, please see the “[Further Reading](#)” section at the end of this chapter.

## Performance in the Architectural Context

What exactly do we mean by *performance* in architecture terms? According to Bass, Clements, and Kazman *Software Architecture in Practice*, performance is “about time and the software system’s ability to meet its timing requirements.”<sup>2</sup> It is also about the management of system resources in the face of particular types of demand to achieve acceptable timing behavior. Performance is typically measured in terms of throughput and latency for both interactive and batch systems.

<sup>2</sup> Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice*, 3rd ed. (Addison-Wesley, 2012), 131.

Performance can also be defined as the ability for a system to achieve its timing requirements, using available resources, under expected full-peak load.<sup>3</sup> In this definition, expected full-peak load could include high transaction volumes, a large number of users, or even additional transactions as a result of a software change. A system is usually defined as a combination of software and computing infrastructure and may include human resources required to operate the software and associated infrastructure.

<sup>3</sup> Definition adapted from the International Organization for Standardization and International Electrotechnical Commission, ISO/IEC 25010:2011(E), *Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models*. <https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en>

Performance and scalability are certainly related, but they are distinct. To illustrate this point, let's consider the checkout process in an e-commerce site. Performance is how quickly the system can process one customer's purchases. Scalability is having multiple queues of customers and being able to add and remove checkout processes as the number of customers changes. Most systems have performance concerns; only systems with significantly variable workloads have scalability concerns. In addition, performance is concerned mainly with throughput and latency (i.e., response time). Although scalability deals with response time, it also is concerned with other aspects such as storage or other computational resource utilization.

Performance degradation, as the workload of a software system increases, may be the first indicator that a system has scalability issues. Several of the performance tactics are also scalability tactics, and because they are discussed in [Chapter 5](#), “[Scalability as an Architectural Concern](#),” we avoid repeating ourselves and refer readers back to [Chapter 5](#) where appropriate in this chapter.

## Forces Affecting Performance

One way to look at performance is to see it as a contention-based model, where the system's performance is determined by its limiting constraints,

such as its operating environment. As long as the system's resource utilization does not exceed its constraints, performance remains roughly linearly predictable, but when resource utilization exceeds one or more constraints, response time increases in a roughly exponential manner.

Architectural decisions decide how these forces interact. Performance is optimized when demand for resources does not cause resource utilization to exceed its constraints. When resource demand overwhelms resource supply, performance degrades exponentially. This has a cost in terms of customer experience and, potentially, market value. On the other hand, when resource supply overwhelms resource demand, the organization has overbought and needlessly increased its cost.

Resource demand is often variable and unpredictable, especially in Web and e-commerce software systems where demand can suddenly increase without warning. Resource supply is harder to scale quickly, as in the case where memory, disk space, and computing power have traditionally been fixed in the very short term and are changeable only through physical upgrades.

Cloud-based architectures, especially serverless architectures,<sup>4</sup> are reducing a lot of these constraints.

<sup>4</sup> See “[Serverless Architectures](#)” section in this chapter.

Performance can be improved by controlling resource demand as well as managing resource supply, using a number of architecture tactics. Those tactics are discussed in the “[Architecting for Performance](#)” section of this chapter.

## Architectural Concerns

When we discuss performance, we are concerned about timing and computational resources, and we need to define how to measure those two variables. It is critical to define clear, realistic, and measurable objectives from our business partners to evaluate the performance of a system. Two groups of measurements are usually monitored for this quality attribute.

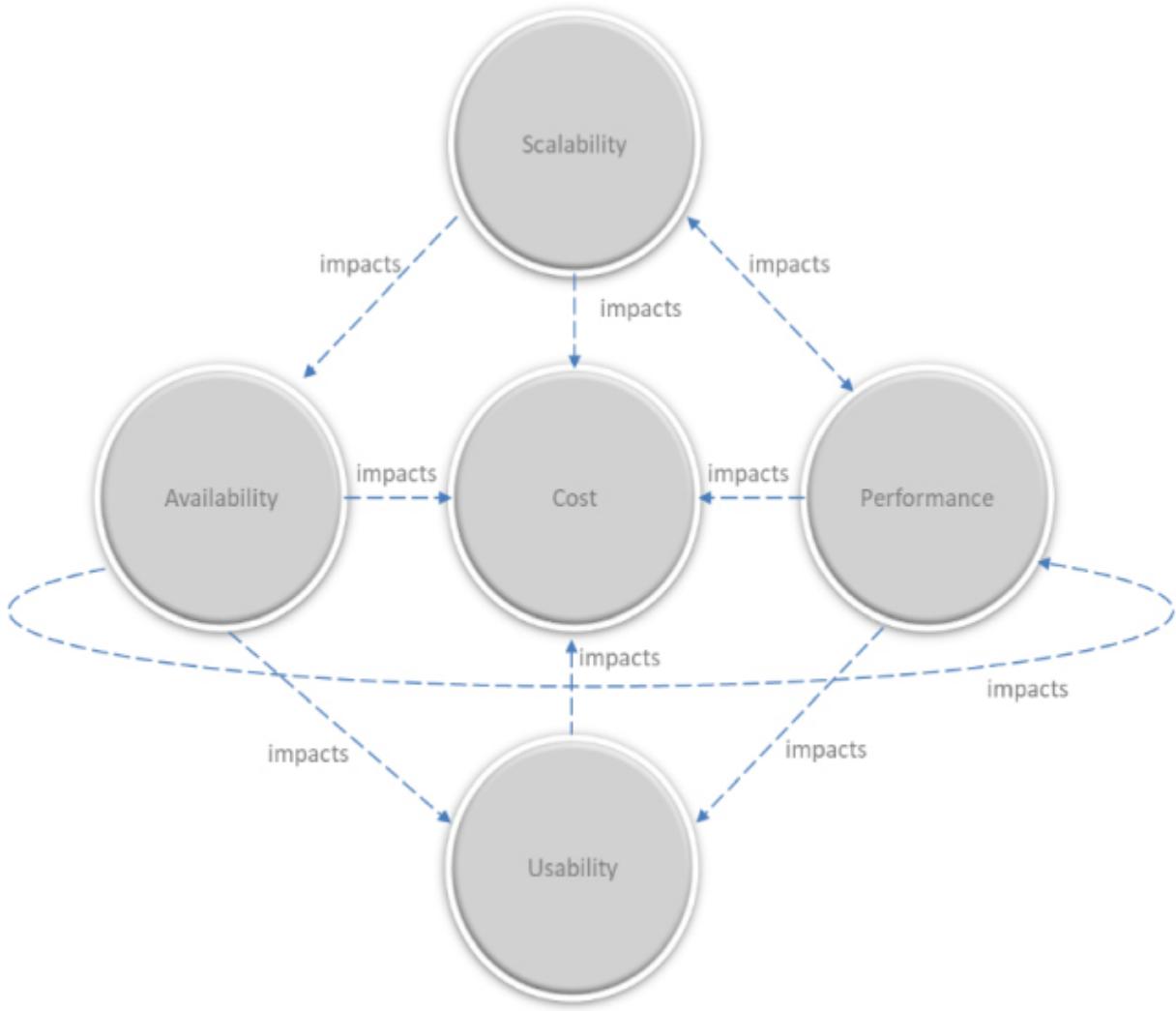
The first group of measurements defines the performance of the system in terms of timings from the end-user viewpoint under various loads (e.g., full-peak load, half-peak load). The requirement may be stated using the end-user viewpoint; however, the measurements should be made at a finer-

grained level (e.g., for each service or computational resource). The software system load is a key component of this measurement set, as most software systems have an acceptable response time under light load. Examples of the measurements included in this group are as follows (please note that others may define these terms differently, which illustrates the need to operationalize definitions using a method such as quality attribute scenarios):

- *Response time/latency*: Length of time it takes for a specified interaction with the system to complete. It is typically measured in fractions of seconds.
- *Turnaround time*: Time taken to complete a batch of tasks. It is typically measured in seconds, minutes, or even hours.
- *Throughput*: Amount of workload a system is capable of handling in a unit time period. It can be measured as a number of batch jobs or tasks per time interval or even by number of simultaneous users that can be handled by the system without any performance issue.

The second group of measurements defines the computational resources used by the software system for the load and assesses whether the planned physical configuration of the software system is sufficient to support the expected usage load plus a safety margin. This measurement can be complemented by stress testing the system (i.e., increasing the load until the performance of the system becomes unacceptable or the system stops working). In addition, performance may be constrained by the end user's computing environment if some components execute on that environment. This would typically happen if the system includes software components (e.g., JavaScript) that execute within browsers running on end-user environments.

As discussed in [Chapter 5](#), performance, scalability, usability, availability, and cost all have significant impacts on each other (see [Figure 6.1](#)).



**Figure 6.1** Performance–usability–scalability–availability–cost relationships

Cost effectiveness is not commonly included in the list of quality attribute requirements for a software product, yet it is almost always a factor that must be considered. In Continuous Architecture, cost effectiveness is an important consideration.

Architectural decisions for performance could have a significant impact on deployment and operations costs and may require tradeoffs between performance and other attributes, such as modifiability. For example, the TFX team uses principle 4, *Architect for change—leverage the “power of small,”* in order to maximize modifiability. Unfortunately, using a lot of smaller services could have a negative impact on performance, as calls

between service instances may be costly in terms of execution time because of the overhead of generating and processing interservice requests. It may be necessary to regroup some of those services into larger services to resolve performance issues.

## Architecting for Performance

In this section, we discuss how the team can architect TFX to meet its performance requirements. We present the impact of emerging trends and talk about some newer performance pitfalls. We review how to architect software systems around performance testing and conclude by discussing modern application and database tactics that can be used to ensure that a software system meets its performance requirements.

### Performance Impact of Emerging Trends

A number of architecture and technology trends have emerged and have become popular over the last few years. Unfortunately, while highly beneficial in some ways, they have also created new performance challenges.

#### *Microservice Architectures*

The first of these trends is the adoption of microservice architectures. [Chapter 5](#) has a detailed discussion of microservice architectures, so we include just a brief summary here. These architectures use loosely coupled components, each with a bounded and cohesive set of responsibilities. Microservices use simple integration constructs such as RESTful APIs. Among the characteristics of microservices, size has turned out to be much less important than loose coupling and doing a few specific things well. To create a coherent, loosely coupled set of services, useful design approaches include using a **Domain-Driven Design (DDD)** and applying the bounded context pattern to determine the boundaries of the services.

Microservices communicate with each other using a lightweight protocol such as HTTP for synchronous communications or a simple message or event bus that follows the smart endpoints and dumb pipes pattern for asynchronous ones. Microservice architectures are widely used, and

supporting tools and processes are widely available. They are based on open source technologies, support most modern languages, and offer both stateless and stateful models.

Unfortunately, excessively distributing an architecture in order to maximize modifiability, by overapplying principle 4, *Architect for change—leverage the “power of small,”* can create a performance pitfall. Calls between too many microservice instances can be costly in terms of overall execution time. Improving performance can be achieved by grouping some of the smaller microservices into larger services, similarly to denormalizing relational databases to improve performance.

## **NoSQL Technology**

The second trend is the growing adoption of NoSQL technology. [Chapter 3](#), “[Data Architecture](#),” presents an overview of NoSQL databases, including a discussion of the main types of NoSQL database technologies, so we include only a brief summary in this section. Traditional relational databases and their performance tactics have limitations when trying to process the workloads generated by the Internet, social media, and e-commerce era. Consequently, companies such as Google, Amazon, Facebook, and Netflix started experimenting with other database architectures. This effort resulted in the emergence of NoSQL databases, which address specific performance challenges.

Selecting the appropriate NoSQL database technology to address a specific performance problem requires a clear understanding of read and write patterns. Using a NoSQL database technology that is inappropriate for the performance issue to be solved will create a performance pitfall. [Table 6.1](#) shows the architecture decision log entry for selecting the database technology for the TFX Good Tracking Service (the full TFX architectural decision log is located in [Appendix A, Table A.2](#)).

**Table 6.1** Decision Log Entry for TFX Good Tracking Database Technology Selection

Name	Description	Options	Rationale
Good Tracking Database	The Good Tracking Database will utilize a document store for its persistence layer.	Option 1, Document store Option 2, Key-value store Option 3, Relational database	Evolving data model requirements Developer ease of use Sufficient to meet known scalability, performance, and availability requirements

Although the team has gone down the path of using a document store, this database technology is known not to scale as efficiently as key–value or wide-column stores for high reads, a shortcoming that would impact performance. The team needs to keep a watch on this. This example shows that there is a close relationship between scalability and performance. It also highlights that architectural decisions are tradeoffs. In this case, the team has opted to focus on the flexibility and ease of use of the database technology.

Another data trend is the growing adoption of big data architectures. This trend is related to the previous one, as big data architectures often use NoSQL database technologies. According to Gorton and Klein,<sup>5</sup> big data systems share the following common requirements:

<sup>5</sup> Ian Gorton and John Klein, “Distribution, Data, Deployment: Software Architecture Convergence in Big Data Systems,” *IEEE Software* 32, no. 3 (2014): 78–85.

- Write-heavy workloads
- Variable request loads
- Computation-intensive analytics
- High availability

Currently available catalogs in the industry do not contain tactics specific to big data systems, but the performance design tactics discussed later in this section can be extended to big data systems.

## ***Public and/or Commercial Clouds***

Finally, the wide adoption of public and/or commercial clouds and the emerging use of serverless architectures have an impact on how architects deal with performance in modern software products. Those two topics are discussed in [Chapter 5](#), so we provide just a brief summary here and cover their impact on performance.

Public and/or commercial clouds provide a number of important capabilities, such as the ability to pay as resources are being used and to rapidly scale when required, especially when containers are being used. They offer the promise of allowing a software system to handle unexpected workloads at an affordable cost without any noticeable disruption in service to the software system's customers and, as such, are powerful mechanisms to help meet performance goals.

However, a common fallacy is that performance is the problem of the cloud provider. Performance issues caused by poor application design are not likely to be addressed by running the software system in a container on a commercial cloud, especially if the original software system design is old and monolithic. Attempting to solve this kind of performance challenge by leveraging a commercial cloud is not likely to be successful and probably won't be cost effective.

Elastic scalability is the preferred approach with cloud computing and works well for virtual machines (VMs) as well as containers.<sup>6</sup> VM scaling is slower than container scaling but still viable for many situations. Elastic scalability could alleviate performance issues when workloads increase, provided that the software system is architected to take advantage of this approach (refer to [Chapter 5](#)) and does not have any performance issues under light load. However, the performance of a software system with an unacceptable response time under light load is not likely to improve by moving that product to a commercial cloud.

<sup>6</sup> For more details on containers, see “Containers at Google.” <https://cloud.google.com/containers>

The location of the data is another important consideration for software systems located in a commercial cloud. Latency issues may occur unless the data is collocated with the application, especially if the data needs to be accessed frequently. For example, accessing databases located within an enterprise's data center from a software system running in a commercial

cloud is likely to create performance issues. Integration patterns between components in the cloud and on-premise not only can cause performance degradation but also can result in significant cost impacts. This is because cloud providers all have a model of charging for egress<sup>7</sup> costs.

<sup>7</sup> Egress is the data that is sent from a cloud to on-premise environments.

## ***Serverless Architectures***

Serverless architectures are discussed in [Chapter 5](#), so we include just a brief summary here. These architectures use a cloud-based model. In this model, the cloud provider manages the computing environment as well as the allocation of resources for running customer-written functions. Software engineers can ignore infrastructure concerns such as provisioning servers, maintaining their software, and operating the servers. However, lack of control over the environment may make performance tuning harder.

One of the advantages of serverless architectures is the ability to run application code from anywhere—for example, on edge servers close to end users—assuming that the cloud provider allows it. This provides an important performance advantage by decreasing latency. It is especially effective when serverless architectures use large commercial vendors with data centers located around the world.

Serverless functions are primarily an event-driven<sup>8</sup> architecture style and, when mixed with more traditional components such as microservices for different parts of a system, can also support a service-based style. Because serverless functions are more granular than microservices, serverless architectures tend to be highly distributed, which may create a performance pitfall similar to the one we discussed for microservices. Performance can be improved by grouping some of the smaller functions into larger ones.

<sup>8</sup> See Wikipedia, “Event-Driven Architecture.” [https://en.wikipedia.org/wiki/Event-driven\\_architecture](https://en.wikipedia.org/wiki/Event-driven_architecture)

In addition, the serverless event-driven model may create design challenges for architects who are not used to this model, and these design challenges may result in performance issues. If not done carefully, utilizing serverless functions can create a modern spaghetti architecture where dependencies between functions get too complex and unmanageable. Ensuring that

appropriate design skills are available within the team is a good approach to mitigate this risk.

## Architecting Applications around Performance Modeling and Testing

How does Continuous Architecture help a team ensure that a system's performance is adequate? For TFX, the team leverages principle 5, *Architect for build, test, deploy, and operate*,<sup>9</sup> and uses a continuous testing approach. This approach is discussed in detail in [Chapter 2, “Architecture in Practice: Essential Activities.”](#) Using this approach, system tests, such as functionality and performance tests, are an integral part of the team's software system deployment pipeline.

In addition to the continuous architecture principles and essential activities discussed in the first two chapters, architects can use various tools to ensure that the performance of a software product meets or exceeds the requirements from its stakeholders. Those tools include a number of performance tactics, which are discussed later in this chapter, as well as performance modeling and performance testing.

Using performance modeling and testing helps the TFX team predict and identify performance pitfalls before the TFX platform is fully built. In addition, analysis of performance testing results enables the team to rapidly modify their design to eliminate performance bottlenecks if necessary.

### ***Performance Modeling***

Performance modeling<sup>9</sup> is initiated with the design of a software system and is part of its development. It includes the following components:

<sup>9</sup> See Bob Wescott, “Modeling,” in *Every Computer Performance Book: How to Avoid and Solve Performance Problems on the Computers You Work With* (CreateSpace Independent Publishing Platform, 2013). Also see “[Further Reading](#)” in this chapter.

- A performance model that provides an estimate of how the software system is likely to perform against different demand factors. Such a model is normally based on the team's prior experience with similar products and technologies. Its purpose is to estimate the performance (specifically the latency) of the components of the software system,

based on the structure of the software system components, the expected request volume, and the characteristics of the production environment. Implementation options for performance models include using a spreadsheet, an open source or commercial modeling tool,<sup>10</sup> or even custom code. You can use anything that allows you to capture your components and the set of performance-related relationships among them. The structure of the software system components used for the model need not be the same as the actual structure of the software system. The team may choose to represent a portion of the system as a black box that comprises multiple services within a model. In these cases, they need to create a mapping between the performance model and each service. The latency allocated to a black box in the performance model should be decomposed and allocated to each of the constituent services.

<sup>10</sup> Discrete event simulation packages (e.g., see [https://en.wikipedia.org/wiki/List\\_of\\_discrete\\_event\\_simulation\\_software](https://en.wikipedia.org/wiki/List_of_discrete_event_simulation_software)) and queuing model packages (e.g., see <http://web2.uwindsor.ca/math/hlynka/qsoft.html>) can be used for this purpose.

- A data capture and analysis process that enables the team to refine the model based on performance tests results. Performance measurements obtained from the performance tests are compared to the model's prediction and used to adjust the model parameters.

Using a performance model and process focuses the TFX team on performance and enables them to make informed architecture decisions based on tradeoffs between performance and other quality attributes, such as modifiability and cost. The architecture decisions are documented in the TFX architectural decision log (see [Appendix A, Table A.2](#)). Using the architectural decision log, design options can be evaluated so that time and resources are not wasted in a design that would create performance bottlenecks. A detailed description of performance modeling is beyond the scope of this book, and the “[Further Reading](#)” section later in this chapter includes references to good materials on this topic, such as Bondi’s *Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice*.

## ***Performance Testing***

The purpose of performance testing is to measure the performance of a software system under normal and expected maximum load conditions. Because TFX will be deployed on a commercial cloud platform, performance testing needs to take place on the same commercial cloud platform, in an environment that is as close as possible to the production environment. An infrastructure-as-code approach to managing TFX’s environments will make this relatively straightforward to achieve as well as ensure reliable environment creation and updates.

As previously discussed, the performance testing process in a continuous delivery environment, such as the one used for TFX, should be fully integrated with software development activities and as automated as possible. The goal is to shift left performance testing activities, identify performance bottlenecks as early as possible, and take corrective action. An additional benefit is that the TFX stakeholders are kept informed of the team’s progress, and they will be aware of problems as they emerge.

Several types of performance tests exist:

- *Normal load testing*: Verifies the behavior of TFX under the expected normal load to ensure that its performance requirements are met. Load testing enables us to measure performance metrics such as response time, responsiveness, turnaround time, throughput, and cloud infrastructure resource utilization levels. Cloud infrastructure resource utilization levels can be used to predict the operational cost of TFX.
- *Expected maximum load testing*: Similar to the normal load testing process, ensures that TFX still meets its performance requirements under expected maximum load.
- *Stress testing*: Evaluates the behavior of TFX when processing loads beyond the expected maximum. This test also uncovers “rare” system issues that occur only under very high load and helps to establish how resilient the system is (which is further discussed in [Chapter 7](#), “[Resilience as an Architectural Concern](#)”).

In practice, the team needs to run a combination of these tests on a regular basis to identify the performance characteristics of TFX. Tests in the category of normal load tests are run more frequently than tests of the other two types.

Finally, it may become unfeasible to run comprehensive performance tests for software products that have very large deployment footprints and/or that manage massive datasets, such as systems used by some social media or e-commerce companies. Examples of testing strategies used by those companies in the production environment include canary testing and a chaos engineering approach with controlled failures, implemented using automation technology such as Netflix’s Simian Army.<sup>11</sup> Fortunately, we do not expect TFX to fall into this category.

<sup>11</sup> See Ben Schmaus, “Deploying the Netflix API” [blog post] (August 2013).  
<https://techblog.netflix.com/2013/08/deploying-netflix-api.html>

Having provided an overview of the impact of continuous delivery on a team’s approach to ensuring that a system’s performance is adequate, we now turn our attention to how to architect for performance, using the TFX case study to provide practical examples and actionable advice.

## Modern Application Performance Tactics

**Table 6.2** lists some of the application architecture tactics that can enhance the performance of a software system such as TFX. These tactics are a subset of the performance tactics described in Bass, Clements, and Kazman’s *Software Architecture in Practice*.<sup>12</sup>

<sup>12</sup> Bass, Clements, and Kazman, *Software Architecture in Practice*, 141.

**Table 6.2** Application Performance Tactics

Control Resource Demand–Related Forces	Manage Resource Supply–Related Forces
Prioritize requests	Increase resources
Reduce overhead	Increase concurrency
Limit rates and resources	Use caching
Increase resource efficiency	

### **Tactics to Control Resource Demand–Related Forces**

The first set of performance tactics affect the demand-related forces in order to generate less demand on the resources needed to handle requests.

## Prioritize Requests

This tactic is an effective way to ensure that the performance of high-priority requests remains acceptable when the request volumes increase, at the expense of the performance of low-priority requests. It is not always feasible, as it depends on the system functions. For TFX, the letter of credit (L/C) issuance process and L/C payment transaction are examples of high-priority requests, while queries (especially those that need to access a significant number of records in the TFX databases) may be processed as lower priority. One option for the TFX team to implement this tactic is to use a synchronous communication model for high-priority requests, while lower-priority requests could be implemented using an asynchronous model with a queuing system. As discussed in [Chapter 5](#), asynchronous interservice interactions do not block (or wait) for the request to return from the service in order to continue. Those requests could be stored in queues, perhaps using a message bus (see [Figure 6.2](#)) until sufficient resources are available to process them.

## Reduce Overhead

As previously mentioned, grouping some of the smaller services into larger services may improve performance, as designs that use a lot of smaller services for modifiability may experience performance issues. Calls between service instances create overhead and may be costly in terms of overall execution time.

Following Domain-Driven Design concepts, the TFX system is designed as a set of loosely coupled components. Although there are some cross-component calls in this design, we see limited opportunity for further consolidation of components. However, let us assume that originally, the Counterparty Service was split into two components: a Counterparty Manager, which deals with the main data about a counterparty, and an Account Manager, which deals with the accounts of the counterparty. This easily could have been a viable alternative. However, the level of cross-component communication created by this approach potentially would have

created performance bottlenecks. If the team had gone down the path of having two services instead of one, they might have opted to consolidate both components into a Counterparty Service to address performance concerns.

Documenting the tradeoffs between performance and modifiability in an architecture decision log (see [Appendix A](#)) is important to ensure that neither of those two quality attributes is sacrificed at the expense of the other.

## **Limit Rates and Resources**

Resource governors and rate limiters, such as the SQL Server Resource Governor<sup>13</sup> and the MuleSoft Anypoint API Manager,<sup>14</sup> are commonly used with databases and APIs to manage workload and resource consumption and thereby improve performance. They ensure that a database or a service is not overloaded by massive requests by throttling those requests. The team has the option to implement an API manager for the entire system and can choose to implement service-specific database governors where relevant.

<sup>13</sup> See Microsoft, “Resource Governor” (December 21, 2020). <https://docs.microsoft.com/en-us/sql/relational-databases/resource-governor/resource-governor?view=sql-server-ver15>

<sup>14</sup> See MuleSoft, “Anypoint API Manager.” <https://www.mulesoft.com/platform/api/manager>

## **Increase Resource Efficiency**

Improving the efficiency of the code used in critical services decreases latency. Unlike the other tactics, which are external to the utilized components, this tactic looks internally into the component. The TFX team schedules periodic code reviews and tests, focusing on the algorithms used and how these algorithms use CPU and memory.

## ***Tactics to Manage Resource Supply–Related Forces***

The second set of performance tactics affect the supply-related forces in order to improve the efficiency and effectiveness of the resources needed to handle requests.

## Increase Resources

Using more powerful infrastructure, such as more powerful servers or a faster network, may improve performance. However, this approach is typically less relevant for cloud-based software systems. Similarly, using additional service instances and database nodes may improve overall latency, assuming that the design of the system supports concurrency. This would work well for a software product such as TFX, which uses a service based architecture.

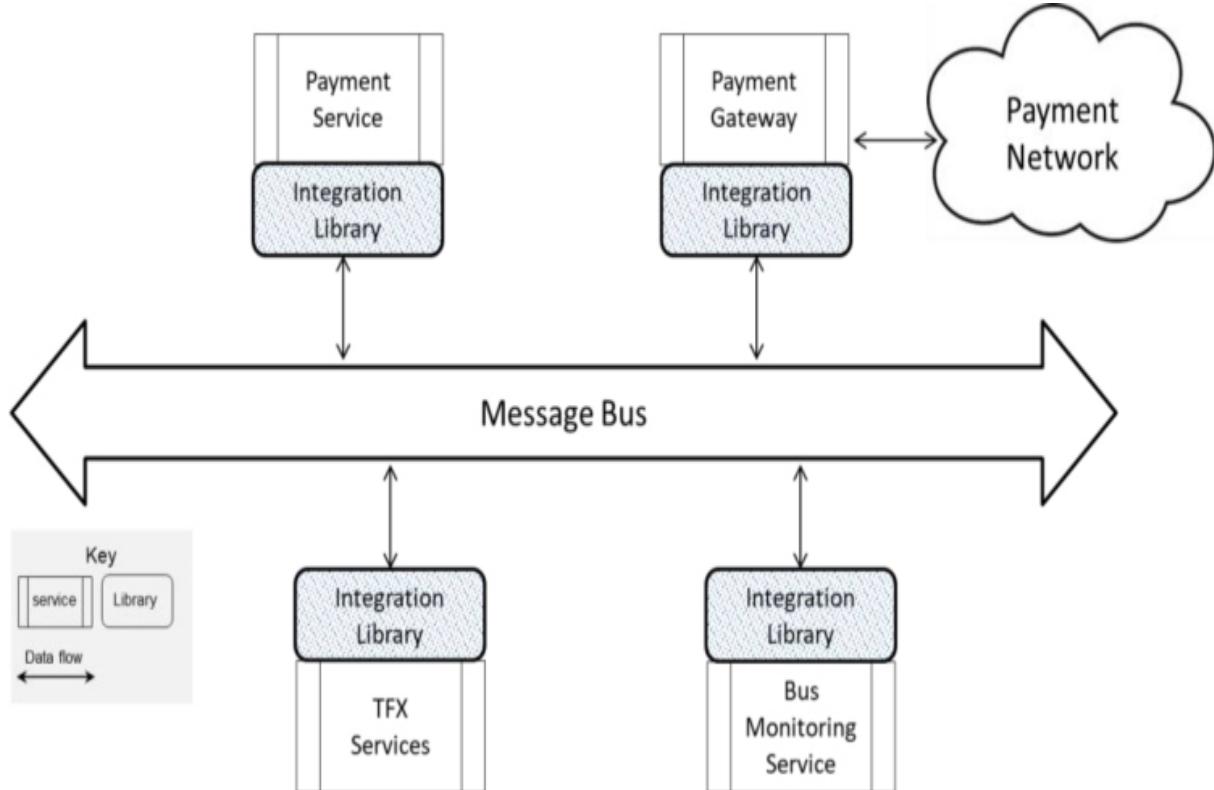
## Increase Concurrency

This tactic involves processing requests in parallel, using techniques such as threads,<sup>15</sup> additional service instances, and additional database nodes. It is also about reducing blocked time (i.e., the time a request needs to wait until the resources it needs becomes available).

<sup>15</sup> Threads are a way for a component to split into two or more concurrently running tasks.

- Processing requests in parallel assumes that the software product is designed to support concurrent processing, and using stateless services is one option to achieve this. It is especially useful for software systems such as TFX, which are architected to run in commercial clouds and take advantage of elastic scalability. Processing requests in parallel requires providing each transaction or each user with its own memory space and resource pools, whether physical or virtual. With small loads, creating a new process for each session/transaction is a simple way to both provide private workspaces and utilize the inherent scaling features of the operating system and the native hardware environment, but creating a process has some overhead, and a large number of processes can be hard to manage. One solution is to use multithreaded programming to handle sessions, but this is a fairly complex approach. Containers and programming frameworks can be used to hide this complexity from software developers who are not experts in multithreading.
- Blocked time can be reduced by implementing asynchronous communications to invoke services that may create a performance bottleneck. As mentioned earlier in this chapter, asynchronous

interservice interactions do not block (or wait) for the request to return from the service in order to continue, thus reducing request blocked time. The TFX team has the option to implement this tactic for services that interface with the TFX Payment Service (see [Figure 6.2](#)).



**Figure 6.2** Implementing asynchronous communications for the TFX Payment Service

## Use Caching

Caching is a powerful tactic for solving performance issues, because it is a method of saving results of a query or calculation for later reuse. Caching tactics are covered in [Chapter 5](#), so only a brief summary of tactics is given here:

- *Database object cache*: This technique is used to fetch the results of a database query and store them in memory. The TFX team has the option to implement this technique for several TFX services. Because early performance testing has shown that some TFX transactions such as an L/C payment may experience some performance challenges, they

consider implementing this type of cache for the Counterparty Manager, the Contract Manager, and the Fees and Commissions Manager components.

- *Application object cache*: This technique caches the results of a service that uses heavy computational resources for later retrieval by a client process. However, the TFX team isn't certain that they need this type of cache, so they decide to apply the third Continuous Architecture principle, “Delay design decisions,” and defer implementing application object cache.
- *Proxy cache*: This technique is used to cache retrieved Web pages on a proxy server<sup>16</sup> so that they can be quickly accessed next time they are requested, either by the same user or by a different user. It may provide some valuable performance benefits at a modest cost, but because the team does not yet have any specific issues that this technique would address, they decide to defer its implementation following principle 3, *Delay design decisions until they are absolutely necessary*.

<sup>16</sup> Proxy servers act as intermediaries for requests from clients within an enterprise that need to access servers external to the enterprise. For more details, see Wikipedia, “Proxy Server.” [https://en.wikipedia.org/wiki/Proxy\\_server](https://en.wikipedia.org/wiki/Proxy_server)

## Modern Database Performance Tactics

Many traditional database performance tactics, such as materialized views, indexes, and data denormalization, have been used by database engineers for decades. The “[Further Reading](#)” section lists some useful materials on this topic. New tactics, such as NoSQL technologies, full-text searches, and MapReduce have emerged more recently to address performance requirements in the Internet era. In addition, the schema on read versus schema on write tradeoff discussed in [Chapter 3](#) can also be viewed as a database performance tactic.

### ***Materialized Views***

Materialized views can be considered as a type of precompute cache. A materialized view is a physical copy of a common set of data that is frequently requested and requires database functions (e.g., joins) that are input/output (I/O) intensive. It is used to ensure that reads for the common

dataset can be returned in a performant manner, with a tradeoff for increased space and reduced update performance. All traditional SQL databases support materialized views.

## ***Indexes***

All databases utilize indexes to access data. Without an index, the only way a database can access the data is to look through each row (if we think traditional SQL) or dataset. Indexes enable the database to avoid this by creating access mechanisms that are much more performant.<sup>17</sup> However, indexes are not a panacea. Creating too many indexes adds additional complexity to the database. Remember that each write operation must update the index and that indexes also take up space.

<sup>17</sup> Detailing how indexing works and related tactics are described in several books—see “[Further Reading](#).” The tactics are specific to the database technology chosen.

## ***Data Denormalization***

Data denormalization is the opposite of data normalization. Whereas data normalization breaks up logical entity data into multiple tables, one per logical entity, data denormalization combines that data within the same table. When data is denormalized, it uses more storage, but it can improve query performance because queries retrieve data from fewer tables.

Denormalization is a good performance tactic in implementations that involve logical entities with high query rates and low update rates.

Updating a denormalized table may involve having to update several rows with the same data, whereas only one update would have been required before denormalization.

Many NoSQL databases use this tactic to optimize performance, using a table-per-query data modeling pattern. This tactic is closely related to materialized views, discussed previously, as the table-per-query pattern creates a materialized view for each query. The next section discusses NoSQL databases in more detail.

## ***NoSQL Technology***

As described earlier, traditional databases and their performance tactics reached limitations in the Internet era. This created the wide adoption of NoSQL databases, which address particular performance challenges. This variety makes it even more critical that we understand our quality attribute requirements (principle 2, *Focus on quality attributes, not on functional requirements*).

---

## Caution

NoSQL technology is a broad topic, and there are few generalizations that hold across the entire category.

As explained in [Chapter 3](#), the NoSQL technology has two motivations: reduce the mismatch between programming language data structures and persistence data structures, and allow CAP and PACELC tradeoffs to be optimized by selecting a storage product that fits your use case. What are CAP and PACELC tradeoffs? According to Gorton and Klein,<sup>18</sup> “Distributed databases have fundamental quality constraints, defined by Brewer’s CAP theorem.<sup>19</sup> . . . A practical interpretation of this theorem is provided by Abadi’s PACELC,<sup>20</sup> which states that if there is a partition (P), a system must trade availability (A) against consistency (C); else (E) in the usual case of no partition, a system must trade latency (L) against consistency (C).” Therefore, CAP and PACELC tradeoffs for NoSQL technology are about performance.

<sup>18</sup> Gorton and Klein, “Distribution, Data, Deployment.”

<sup>19</sup> Eric Brewer, “CAP Twelve Years Later: How the ‘Rules’ Have Changed,” *Computer* 45, no. 2 (2012): 23–29.

<sup>20</sup> David Abadi, “Consistency Tradeoffs in Modern Distributed Database System Design: CAP Is Only Part of the Story,” *Computer* 45, no. 2 (2012): 37–42.

To select the correct NoSQL database technology that addresses your performance problems requires a clear understanding of your read and write patterns. For example, wide-column databases perform well with high numbers of writes but struggle with complex query operations. The problem is that in order to be fast for insert and retrieval, those databases have

limited query engines restricting them in the support of complex queries that can be used with them.

NoSQL technology increases the options available for addressing performance concerns. However, we want to highlight that a database technology choice is an expensive decision to change for two main reasons. First, particularly with NoSQL databases, the architecture of the software system is influenced by the database technology. Second, any database change requires a data migration effort. This can be a challenging exercise that not only deals with managing schema differences but requires affirmation that the data is complete and that the business meaning is not altered. If you are interested in a structured approach to evaluating different NoSQL database technologies against quality attributes, LEAP4PD, provided by Carnegie Mellon's Software Engineering Institute is a good reference.<sup>21</sup>

<sup>21</sup> <https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=426058>

## ***Full-Text Search***

Full-text searches are not a new requirement, but they have become more and more prevalent in the internet and ecommerce era. This type of search refers to tactics for searching documents in a full-text database, without using metadata or parts of the original texts such as titles or abstracts stored separately from documents in a database.

Let us look at a potential area where full-text search can be implemented in the TFX system. The team responsible for the Document Manager has chosen the option to use a document database. This decision was driven mainly by the fact that such a model supports the natural structure of documents and can be easily extensible if required. A key performance-related quality attribute was based on full-text searches conducted by end users:

- *Scenario 1 stimulus:* An analyst at an importer bank conducts a search across all L/Cs to determine where a particular individual from a client is referenced in the documentation.
- *Scenario 1 response:* The TFX system provides search results in an online manner, including recommendations for partially completed

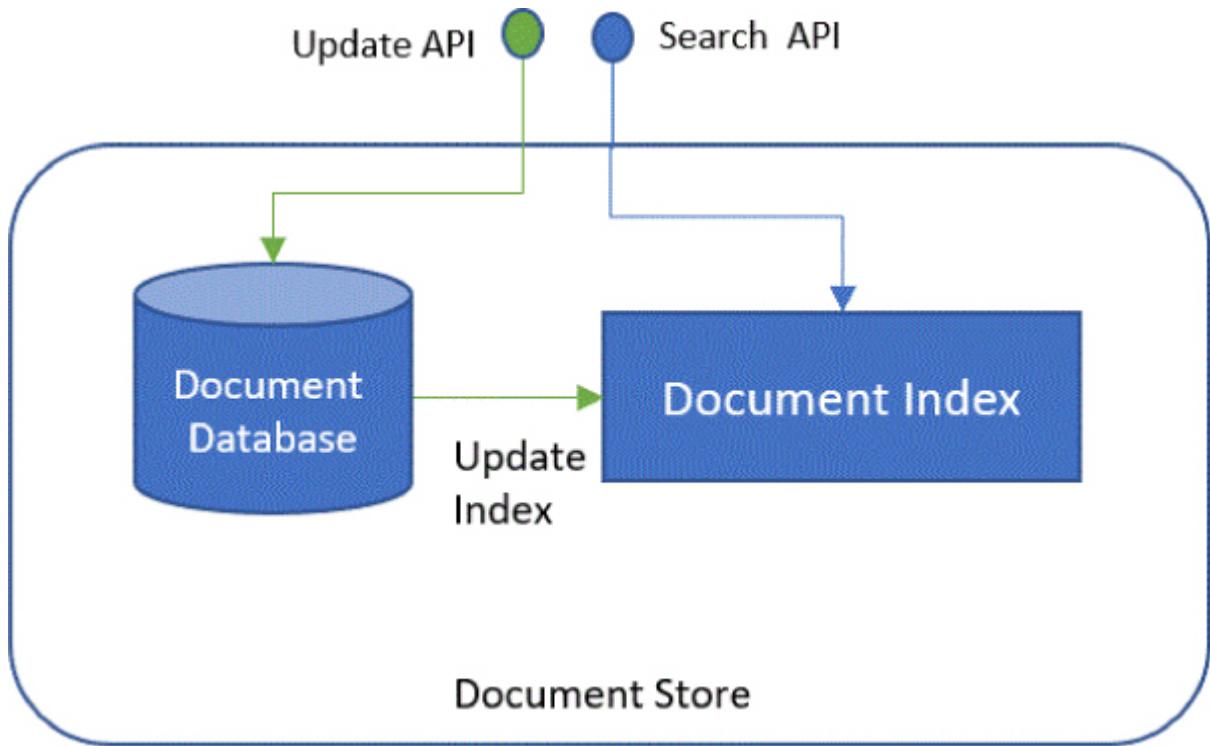
entries as the analyst types.

- *Scenario 1 measurement:* The response time for this search is in real time as the end user types.

Although the underlying database provides search capabilities, it cannot meet the required measurement criteria. The team is aware that several full-text engines exist in the marketplace, the most common ones being Solr and ElasticSearch, which are based on the Lucene search engine. After a few failed attempts at configuring the document database to meet the performance criteria, the team has the option to utilize a full-text search engine as part of the overall solution. Utilization of such a search engine would mean that the document store and search engine are used together. The document store is responsible for managing the documents, and the search engine is responsible for replying to search queries from the UI service (see [Figure 6.3](#)).

Full-text search engines are becoming more prevalent and are commonly used with NoSQL document stores to provide additional capabilities. Many types of NoSQL stores don't typically integrate full-text search, but there are integrations of full-text search with document databases and key-value stores (e.g., ElasticSearch with MongoDB or DynamoDB). Logically, the distinction between a key-value and document store is that a key-value store just has a key and a blob, whereas a document store has a key, some metadata, and a blob. In addition, a document store knows it is storing a “document” and provides support for looking “inside” the document in certain formats, typically XML and JSON. Document stores are better suited to large blobs, but the crossover point depends on the specific engines being used. You can use full-text search on the blob in either case.

For TFX, the team has the option to index each document that is stored in the database to make it available as part of the search engine.



**Figure 6.3** *Search engine extension of document store*

## **MapReduce**

Another common tactic for improving performance (particularly, throughput) is to parallelize activities that can be performed over large datasets. The most common such approach is MapReduce, which gained widespread adoption as part of the Hadoop ecosystem<sup>22</sup> used for big data systems. At a high level, MapReduce is a programming model for large-scale data processing that does not require the programmer to write parallel code. Other distributed processing systems for big data workloads, such as Apache Spark,<sup>23</sup> use programming models that are different from MapReduce but are solving fundamentally the same problem of dealing with large, unstructured datasets.

<sup>22</sup> Massively parallel processing (MPP) databases employ similar parallelization techniques for analytical processing and predate Hadoop.

<sup>23</sup> See “Apache Spark.” <https://spark.apache.org>

MapReduce works by parallelizing calculations on subsets of the entire dataset (map) and then combining the results (reduce). One of the key

principles of MapReduce is that the data is not copied to an execution engine; rather, the calculation instruction code is copied to where the dataset resides on the file system (Hadoop distributed file system [HDFS] in the case of Hadoop). The two key performance benefits of MapReduce are that it implements parallel processing by splitting calculations among multiple nodes, which process them simultaneously, and it collocates data and processing by distributing the data to the nodes.

There are, however, some performance concerns with MapReduce. Network latency can be an issue when a large number of nodes is used. Query execution time can often be orders of magnitude higher than with modern database management systems. A significant amount of execution time is spent in setup, including task initialization, scheduling, and coordination. As with many big data implementations, performance depends on the nature of the software system and is affected by design and implementation choices.

## Achieving Performance for TFX

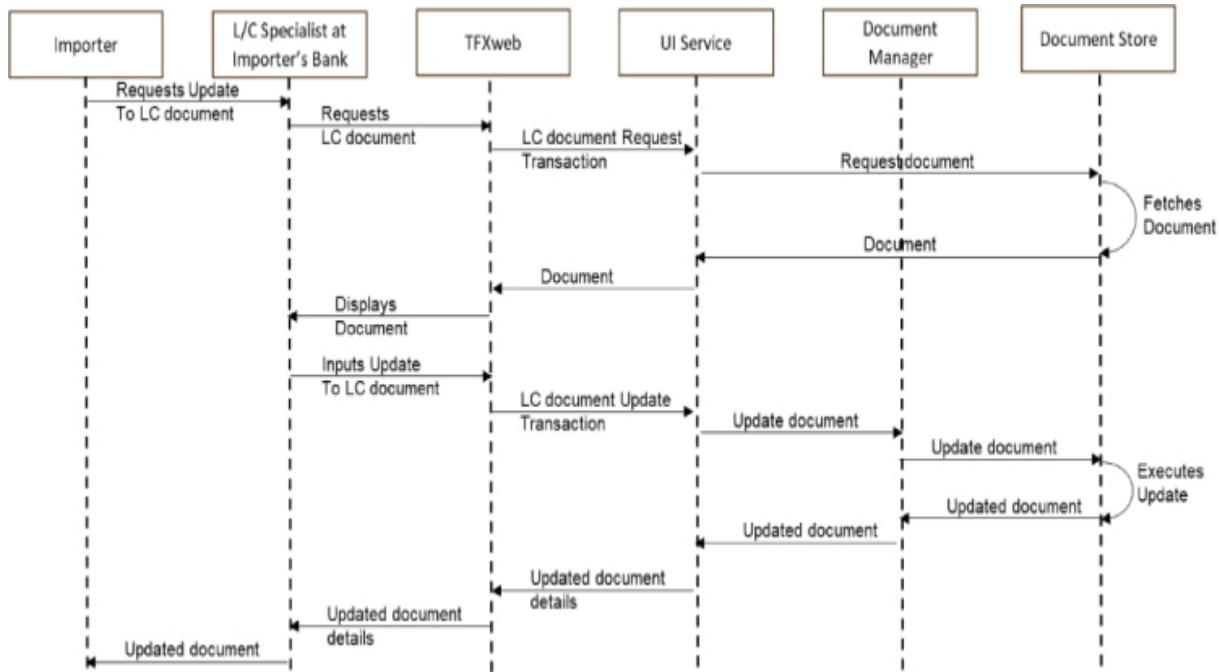
### *TFX Performance Requirements and Tactics*

Capturing and documenting TFX performance requirements seems to be a straightforward process, at least initially. As discussed, quality attribute requirements need to be specified in terms of stimulus, response, and measurement. Let us look at two examples of performance requirements that the TFX stakeholders could provide to the team.<sup>24</sup>

<sup>24</sup> See “Building the Quality Attributes Utility Tree” in Chapter 2, “Architecture in Practice: Essential Activities” for a description of the structure of a scenario.

- *Scenario 1 stimulus:* An L/C specialist at the importer’s bank needs to change a list of goods and their description in a document before an L/C can be issued.
- *Scenario 1 response:* The TFX document management service updates the document as expected.
- *Scenario 1 measurement:* The response time for this transaction does not exceed 3 seconds at any time.

Figure 6.4 shows a sequence diagram for scenario 1.



**Figure 6.4 Scenario 1 sequence diagram**

- *Scenario 2 stimulus:* A letter of credit specialist at the importer’s bank processes a payment request from the exporter’s bank following acceptance of documents from the exporter.
- *Scenario 2 response:* The TFX payment service sends the payment to the exporter’s bank, using the Payment Gateway, and debits the importer’s account.
- *Scenario 2 measurement:* The end-to-end payment transaction time does not exceed 5 seconds.

There is a significant risk associated with meeting these requirements. Meeting those goals will depend on the future load of the system, and, as noted in [Chapter 5](#), we do not have good estimates for how the TFX load is expected to evolve.

The TFX design for performance must be based on facts and educated guesses, given the unknown related to the future load of the platform. A higher workload than expected would turn the team’s initial architecture decisions into performance roadblocks.

Given the uncertainty around future loads, the team follows principle 3, *Delay design decisions until they are absolutely necessary*, and decides not to overarchitect for a future load that may not materialize. They use the volume statistics and projections from our client for their L/C business for the current and next two years in order to create a performance model and design for performance.

We don't have space to consider all the work that the team would need to do to ensure that TFX meets its performance objectives, but we briefly recapitulate some of the tactics they are planning to use for this purpose (see [Table 6.3](#)). Those tactics were discussed in detail earlier.

**Table 6.3** Sample TFX Performance Tactics

Tactic	TFX Implementation
Prioritize requests	Use a synchronous communication model for high-priority requests such as L/C payment transactions. Use an asynchronous model with a queuing system for lower-priority requests such as high-volume queries
Reduce overhead	Where possible, group smaller services into larger ones. A hypothetical example we gave earlier was to merge the Counterparty and Account Manager services into the Counterparty Manager service.
Increase resource efficiency	Schedule periodic code reviews and tests, focusing on the algorithms used and how these algorithms use CPU and memory.
Increase concurrency	Reduce blocked time by using a message bus to interface TFX services with the TFX Payment service.
Use caching	Consider using database object cache for the Counterparty Manager, the Contract Manager, and the Fees and Commissions Manager components.
Use NoSQL technology	Use a document database for the Document Manager and a key-value database for the Payment Service.

## ***Measuring Performance***

One of the team's most important tasks in achieving performance for TFX is to include the necessary architecture components in the design to measure performance and identify bottlenecks. We expect TFX users to occasionally experience response time degradation as workload increases. We need an

approach to identify service slowdowns and failures and to remedy them rapidly.

As we discuss in [Chapters 5](#) and [7](#), the team is implementing a comprehensive monitoring strategy for TFX. This strategy defines which events should be monitored, which tools to use, and what action to take should an event occur. Logging and metrics are a key component of that strategy.

Capturing information to monitor performance should not be an afterthought added after all the TFX components have been designed and implemented. Leveraging principle 5, *Architect for build, test, deploy, and operate*, the team ensures that every TFX component is designed to log useful performance monitoring information, such as execution time and usage metrics in a structured format (see [Figure 6.5](#)).



**Figure 6.5** TFX Monitoring architecture overview

Capturing performance information in logs is essential but not sufficient. As shown in [Figure 6.5](#), logs need to be forwarded to a centralized location in order to be consolidated and analyzed to provide useful information. Log aggregation is part of the overall performance management process and must consolidate different log formats coming from different sources. Being able to aggregate logs in a centralized location with near real-time access is key to quickly troubleshooting performance problems as well as performing analysis. This enables us to better understand the performance profile of each TFX service and quickly identify the source of potential performance issues.

## Potential Performance Bottlenecks

As stated earlier, there is a tradeoff between performance and modifiability. An architecture with a large number of small services increases modifiability but could experience performance issues. It may be necessary to group some of those services into larger services to resolve latency issues.

The team is already leveraging a standard library of software components to handle interservice interfaces in order to be able to quickly address potential performance bottlenecks. This library provides timing information on interservice communications, which can be used to identify significant latencies in such communications.

The data tier of the architecture could become another performance bottleneck as the TFX workload increases. This is related to the PACELC tradeoff discussed earlier.<sup>25</sup> As the databases of the TFX services increase in size, it may become necessary to distribute data across multiple database instances in order to keep performance at an acceptable level, leading to replication of data across database instances, which may increase complexity. Fortunately, the TFX design is based on a set of services organized around business domains, based on Domain-Driven Design principles.<sup>26</sup> This results in TFX data being aligned to services and organized around business domains. This design enables the data to be well partitioned across a series of loosely coupled databases, keeps replication needs to a minimum, decreases complexity, and optimizes performance.

<sup>25</sup> Also see the discussion on replication and partitioning strategies in [Chapter 3, “Data Architecture.”](#)

<sup>26</sup> For more information on Domain-Driven Design, please see Vaughn Vernon, *Implementing Domain-Driven Design* (Addison-Wesley, 2013).

Increasing workload size may cause rare conditions, such as exceeding hardcoded limits, to become more common. As a result, the system is more likely to experience performance edge cases when it approaches its limits. Those conditions need to be identified using stress testing (see “[Architecting Applications around Performance Modeling and Testing](#)” earlier in this chapter).

Finally, performance may be constrained by the end-user computing environment if some components of the system (such as JavaScript components) need to execute in that environment. This creates some additional challenges for performance testing the system, as it needs to include tests that use a range of end-user computing devices.

## Summary

This chapter discussed performance in the context of Continuous Architecture. Unlike scalability, performance has traditionally been at the top of the list of quality attributes for an architecture because poor performance may have a serious impact on the usability of the software system.

This chapter first discussed performance as an architectural concern. It provided a definition of performance and discussed the forces affecting performance as well as the architectural concerns.

The second part of this chapter dealt with architecting for performance. It discussed the performance impact of emerging trends, including microservice architectures, NoSQL technology, public and/or commercial clouds, and serverless architectures. It also discussed how to architect software systems around performance modeling and testing. It then presented some modern application performance tactics as well as some modern database performance tactics such as using NoSQL technology, full-text search, and MapReduce. It concluded with a discussion on how to achieve performance for our test case, TFX, including how to measure performance and potential performance bottlenecks.

As we pointed out in [Chapter 5](#), performance is distinct from scalability. Scalability is the property of a system to handle an increased (or decreased) workload by increasing (or decreasing) the cost of the system, whereas performance is “about time and the software system’s ability to meet its timing requirements.”<sup>27</sup> Also remember that performance, scalability, resilience, usability, and cost are tightly related and that performance cannot be optimized independently of the other quality attributes.

<sup>27</sup> Bass, Clements, and Kazman, *Software Architecture in Practice*.

Many proven architectural tactics can be used to ensure that modern systems provide acceptable performance, and we have presented some of the key ones in this chapter. Scenarios are an excellent way to document performance requirements and help the team create software systems that meet their performance objectives.

Modern software systems need to be designed around performance modeling and testing. In a continuous delivery model, performance tests are an integral part of the software system deployment pipeline.

Finally, to provide acceptable performance, the architecture of a modern software system needs to include mechanisms to capture accurate information to monitor its performance and measure it. This enables software practitioners to better understand the performance profile of each component and quickly identify the source of potential performance issues.

The next chapter discusses resilience in the Continuous Architecture context.

## Further Reading

We are not aware of a definitive book on performance as an architectural concern, but we have found several books and websites helpful.

- Michael T. Nygard's *Release It!: Design and Deploy Production-Ready Software* (Pragmatic Bookshelf, 2018) is primarily about resilience but contains a lot of information that is relevant to performance.
- *Systems Performance: Enterprise and the Cloud* (Pearson Education, 2013) by Brendan Gregg is a very good performance engineering book.
- *Web Performance in Action: Building Faster Web Pages* (Manning Publications, 2017) by Jeremy Wagner is much focused on the Web browser and has a lot of useful information for developers who are dealing with a Web app UI performance problem.
- André B. Bondi's *Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice* (Pearson Education, 2015) is a real “performance for performance engineers” book, taking in foundations,

modeling, testing, requirements, and more. The author is a career performance engineer in research and practice, so we think it's quite reliable.

- Todd DeCappa and Shane Evans, in *Effective Performance Engineering* (O'Reilly Media, 2016) offer a much simpler and shorter book that provides a brief introduction to performance engineering.
- In *Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems* (O'Reilly Media, 2017), Martin Kleppmann discusses reliability, scalability, and performance for data-oriented applications. It is a classic text of its type.
- Bob Wescott's *Every Computer Performance Book: How to Avoid and Solve Performance Problems on the Computers You Work With* (CreateSpace Independent Publishing Platform, 2013) is a short, sometimes humorous book on how to deal with performance challenges.
- Keith Smith's *Fundamentals of Performance Engineering: You Can't Spell Firefighter without IT* (HyPerfomix Press, 2007) is a good introduction to most aspects of IT system performance, covering the culture, organization, and technical pieces in a conversational, easy-to-read style.
- Henry H. Liu's *Software Performance and Scalability: A Quantitative Approach, Vol. 7* (Wiley, 2011) covers enterprise applications performance and provides both the analytical and measurement aspects of performance and scalability.
- Another great resource is Connie Smith's perfeng.com site. She was a pioneer of performance engineering practice in the 1990s and 2000s. Her book with Lloyd Smith, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software* (Addison-Wesley, 2001) was a standard practitioner text for a long time but has been out of print for some years, but you can still find it.<sup>28</sup>
- Trevor Warren's “SPE BoK”<sup>29</sup> has a lot of good information on software performance engineering.

- For readers who are interested in a structured approach to evaluating different NoSQL database technologies against quality attributes, LEAP4PD,<sup>30</sup> provided by the Software Engineering Institute, is a good reference.

<sup>28</sup> For example, <https://www.amazon.com/exec/obidos/ASIN/0201722291/performanceen-20>

<sup>29</sup> <https://tangowhisky37.github.io/PracticalPerformanceAnalyst>

<sup>30</sup> <https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=426058>

We do not know of many product-independent books on database performance and tuning. There are lots of database-specific books on Oracle, DB/2, MySQL, Cassandra, MongoDB, but not so many on the general principles.

- Dennis Shasha and Philippe Bonnet's *Database Tuning: Principles, Experiments, and Troubleshooting Techniques* (Elsevier, 2002) provides a good explanation of the generic fundamentals of relational database management system performance and tuning and explains the process well.
- *Database Design for Mere Mortals: A Hands-on Guide to Relational Database Design* (Pearson Education, 2013) by Michael James Hernandez is a simpler guide on relational database design. It provides good information at the logical level of design but does not deal with physical design aspects.
- For readers interested in learning more about how databases work internally, Alex Petrov's *Database Internals* (O'Reilly Media, 2019) is a useful book written by a Cassandra committer.
- Bill Karwin's *SQL Antipatterns: Avoiding the Pitfalls of Database Programming* (Pragmatic Bookshelf, 2010) is not a book focused on database performance and tuning, but it includes many SQL antipatterns, which are performance related.
- Finally, readers interested in further investigating performance modeling will find a good piece of research, experimenting with UML and the MARTE profile, in Dorina C. Petriu, Mohammad Alhaj, and Rasha Tawhid's "Software Performance Modeling," in Marco

Bernardo and Valérie Issarny (Eds.), *Formal Methods for the Design of Computer, Communication and Software Systems* (Springer, 2012), pp. 219–262.

# Chapter 7. Resilience as an Architectural Concern

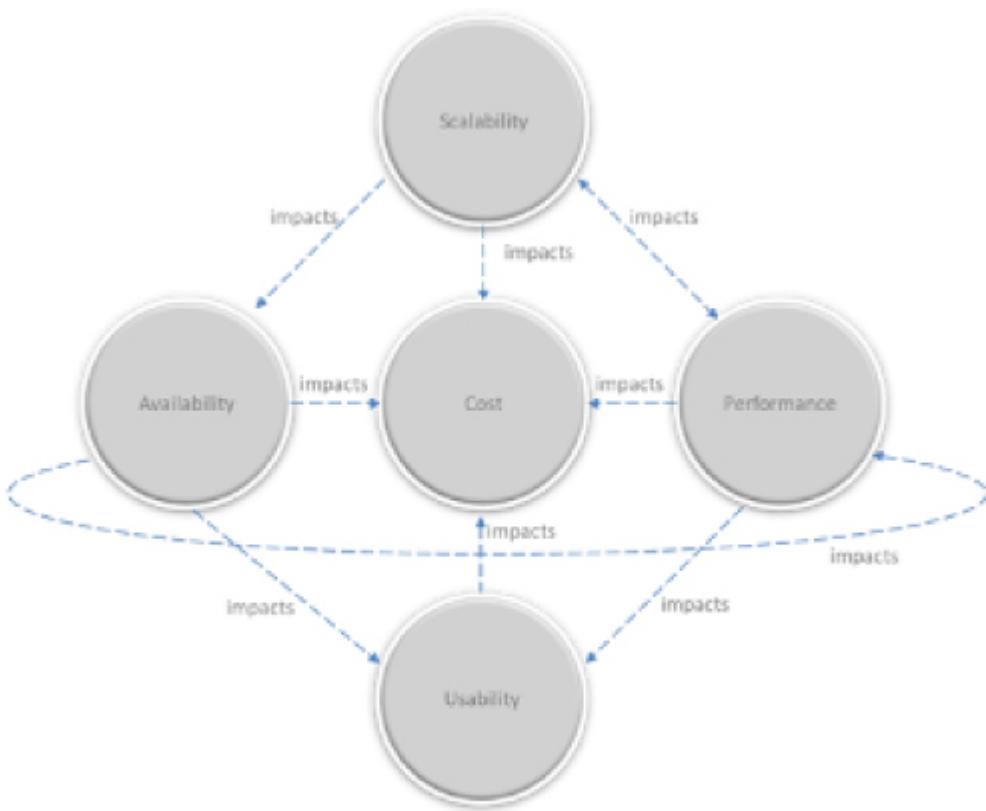
*A system is resilient if it can . . . sustain required operations under both expected and unexpected conditions.*

—Erik Hollnagel

In recent years, we have seen a dramatic shift in the expectations of end users in relation to the availability of the computing systems that they use. Because they have become accustomed to the always-on reliability of Internet services, such as Google Maps, CNN, and the BBC, they now expect this sort of reliability from the systems that we build for them, whether they are providing Internet-facing services or are private systems within our organizations.

To meet this expectation, we have had to update our thinking and techniques from the traditional high-availability approach, where we accepted a certain amount of planned and unplanned downtime, to approaches that aim for no downtime and can provide at least minimal service in the face of almost any problem.

As said in earlier chapters, architectural concerns are interrelated, and resilience is not an exception to this rule. Resilience and its architectural tactics overlap significantly with other quality attributes, including usability, performance, and scalability, influencing the cost of the system as well, as illustrated by [Figure 7.1](#).



**Figure 7.1 Scalability–performance–availability–usability cost relationships**

This chapter considers resilience as a distinct concern, but it refers to [Chapter 5, “Scalability as an Architectural Concern,”](#) and [Chapter 6, “Performance as an Architectural Concern,”](#) given the close relationship among these quality attributes.

Resilience is an important quality attribute for the team in our case study, developing the Trade Finance eXchange (TFX) system. TFX is likely to support the business of a single bank initially, but as a global organization, the bank will have an expectation that the system is available continually, although it probably can accept short periods of unavailability in the early stages of adoption of the platform. However, the ambition is for TFX to be a key platform for letters of credit (L/Cs) for major financial institutions, so in the medium to long term, the team knows that there will be a very low tolerance for failures or outages. TFX’s users will expect it to be an always-on system before they are prepared to use it as a key part of such critical business processes. Therefore, the team need to be thinking now about how

they will achieve resilience without major changes to their architecture when the system gains widespread use.

## Resilience in an Architectural Context

We've noticed that people tend to use many resiliency-related terms interchangeably, which is confusing. Therefore, let's define some terminology so that our meaning is clear.

Two commonly used terms are *fault* and *failure*, which are often used interchangeably but refer to subtly different concepts. A fault is an accidental condition that, if encountered, may cause the system or a system component to fail to perform as required. A failure is when a system or system component deviates from its required behavior. So, a fault is something that has gone wrong and that may cause a failure to occur.<sup>1</sup>

<sup>1</sup> Lorraine Fesq (Ed.), *Fault Management Handbook*, NASA-HDBK-1002 (National Aeronautics and Space Administration, 2011).

What do we mean by *availability*, *reliability*, *high availability*, and *resilience*?

- *Availability* is a measurable quality attribute of a system, defined as the ratio of the time when the system *is* available to the overall time when it *could have been* available. For example, if a system suffers a failure that makes it unavailable for 12 minutes in a week, then the system has exhibited  $1 - (12 \div 60 \times 24 \times 7) = 0.9988$ , or 99.88 percent availability for that week. The complication with availability is that what it means to be "available" is context dependent (and perhaps a matter of opinion too). This is why, in common with other quality attributes, it is important to use concrete scenarios to show what they mean in a particular context. We discuss this point further with examples later in the chapter.
- *Reliability* builds on availability and adds the constraint of correct operation, not just availability. A widely used definition of the reliability of software is "the probability of failure-free software operation for a specified period of time in a specified environment."<sup>2</sup> To be reliable, then, a system needs to be available, but it is possible to

have a highly available system that is not highly reliable, as it may be available but not operating correctly.

Availability and reliability are outcomes, which we can observe from a real system, and they can be achieved in different ways.

- Historically, enterprise systems tended to achieve availability through the use of so-called **high-availability** technology, primarily **clustering**, such as application clusters and clustered databases, and replication, such as cross-site **database replication**. If things went wrong on one server node, the workload was migrated to a surviving node in the cluster. If an entire cluster malfunctioned, then the workload was failed over to a standby cluster that had (most of) the data replicated into it. This approach to availability tries to push error handling and the availability mechanisms out of the application software into complex middleware and database technologies, such as clustered database and application servers and database replication mechanisms. The snag with this approach is that, while convenient for the application developer, the availability mechanisms are quite complex and difficult to use and, in many cases, expensive. Some failover processes are quite involved and require a period of unavailability while they complete. These technologies were also designed for monolithic on-premise systems and are much less well suited to microservice-based, highly distributed, cloud-hosted systems.

However, in recent years, we have seen a move, led by the cloud-based Internet giants such as Amazon, Google, Facebook, and Netflix, from *high availability* to *resilience* in order to achieve availability.

- **Resilience** takes a different approach to achieving reliability. It requires that each part of the system be responsible for contributing to the system's availability by adapting its behavior when faults occur—for example, by recognizing when faults occur, tolerating latency, retrying requests, automatically restarting processes, limiting error propagation, and so on. Architects and software engineers need to be much more aware of the mechanisms required to achieve availability than they were when using high-availability technologies. When done well, however, the resulting systems are more resistant to faults and failures

and can adapt much more flexibly to problems that emerge during their operation.

<sup>2</sup> Michael R. Lyu, *Handbook of Software Reliability Engineering* (IEEE Computer Society Press/McGraw Hill, 1996).

## What Changed: The Inevitability of Failure

*Before* the cloud-computing era, most of our systems were made up of relatively few monolithic pieces and deployed into quite simple, static environments. As already mentioned, we typically ensured the availability of these systems using high-availability technologies such as clusters and replication with failover processes from the failed environment to an idle standby environment.

In recent years, high-availability approaches have become significantly less effective for many systems as those systems have become increasingly larger, comprise many more runtime components, and are deployed into cloud environments. Simple statistical probability tells us that the greater the number of parts, the higher the chances of a malfunction at any point in time, so faults are more common. In general, however, traditional high-availability technology assumes relatively infrequent failures, and so the recovery process is relatively slow. In addition, most of these technologies are not designed for cloud environments or to handle very large numbers of small runtime components; instead, they are designed and optimized for systems with a small number of monolithic components, running on hardware under our control. These factors mean that traditional high-availability failover approaches do not work well for many new applications.

The solution to this problem may sound odd: you just accept that things fail, and you design your system to continue operation and provide service even when some part of it is malfunctioning. As mentioned, this is defined as resilience, and it involves dealing with faults and failures throughout our architecture rather than following the traditional high-availability approach of trying to externalize it to specialist middleware and ignoring it elsewhere.

## Reliability in the Face of Failure

If we accept that our systems are going to suffer faults and experience unexpected environmental conditions but must continue to offer service, we need to design them to prevent failure when unexpected things happen. Classical resilience engineering<sup>3</sup> teaches us that there are four aspects to prevent failure:

<sup>3</sup> Jean Pariès, John Wreathall, David Woods, and Erik Hollnagle, *Resilience Engineering in Practice: A Guidebook* (CRC Press, 2013).

- *Knowing what to do*, that is, knowing which automated mechanisms and manual processes are needed and where to allow us to continue providing service, such as restarting, failing over, shedding load, adding capacity, and so on.
- *Knowing what to look for* so that we can monitor our system and its operating environment to recognize situations that are, or could become, a threat to its availability.
- *Knowing what has happened* so that we can learn from experience what has succeeded and what has failed and use this information to support the previous point—knowing what to look for—and improve our system.
- *Knowing what to expect*, using both backward-looking information and forward-looking predictions to anticipate possible problems in the operating environment.

All four aspects of resilience need to be considered in the context of the technology (our hardware and software), the processes we use to operate the system, and the people who are responsible for its operation, as those three elements combine to form the system at runtime.

## The Business Context

Traditionally, people talk about system availability in terms of the “number of 9s” that are required, referring to the percentage availability where 99 percent (two nines) means about 14 minutes unavailability per day, 99.99 percent means about 8.5 seconds per day, and 99.999 percent (the fabled five nines) means less than a second of unavailability per day.

We have observed three significant problems with the common use of these measures. First, in most situations, it is assumed that more is better, and every business situation claims to need five nines availability, which is expensive and complex to achieve. Second, the practical difference between most of these levels is close to zero for most businesses (e.g., the business impact of 86 seconds [99.9%] and 1 second [99.999%] unavailability in a day is often not materially different). Third, these measures don't consider the timing of the unavailability. For most customers, there is a huge difference in having a system outage for 1 minute in a retail system during the Christmas shopping season and a 20-minute outage of the same system early on a Sunday morning. This is why we have found the number of nines to be less meaningful than most people assume.

Our observation is that, in most businesses today, there are two sorts of system: those that need to be available just about all of the time (e.g., a public-facing API gateway for the TFX inbound transactions) and those for which no or minimal service can be tolerated for a significant period of an hour or two without major business impact (e.g., TFX's reporting components).

Knowing your business context allows you to decide which category your system is in and design its resilience characteristics accordingly.

## **MTTR, Not (Just) MTBF**

Moving our mindset from traditional availability techniques to resilience involves changing how we think about failure and recovery too. Two key metrics are used when considering system availability, mean time between failures (MTBF) and mean time to recover (MTTR), the former being the average amount of time between failures, either actual or estimated, and the latter being the average amount of time taken to restore service when a failure occurs.

Both MTBF and MTTR are more complex than they appear at first glance. What do we classify as a failure? What do we mean by recovery? If we have a system failure but manage to partly hide it from end users, is this still a failure? If we have a system failure and manage to restore our e-commerce store within a minute or so, but can't process a credit card transaction for 20 minutes (perhaps meaning that we accept orders but

process them later when we can validate payment), was our recovery time 1 minute or 20 minutes? It is important that we think clearly about what failure and recovery mean for our system so that we can design a solution that provides our business with resiliency.

Historically, we tended to achieve reliability by focusing on MTBF and using high-availability technology to mask faults and keep the system in service, thereby avoiding the need to recover. The tradeoff tended to be that when we did have an outage, it was complex and time consuming to recover from.

An insight from some of the early Internet firms, notably stated by Etsy's former chief technology officer John Allspaw, was that "TTR is more important than TBF (for most types of F)."⁴ This means that when creating a resilient system, the time to recover is at least as important as the time between failures and is probably easier to optimize for. What these companies found was that, in practice, minimizing the time to recover resulted in higher availability than maximizing time between failures. The reason for this appears to be that, because they faced a very unpredictable environment, problems were inevitable and out of their control. Therefore, they could control only the time taken to recover from problems, not the number of them. When it is possible to recover very quickly from most failures, the failures become much less important, and so their frequency becomes much less significant.

<sup>4</sup> <https://www.kitchensoap.com/2010/11/07/mttr-mtbf-for-most-types-of-f>

Both approaches are still perfectly valid, but they imply different tradeoffs and need different architectural tactics to be used. It is important, then, that we are intentional in which we optimize for—time between failures or time to recover. It is difficult to optimize for both.

In our example, the TFX team has prioritized fast recovery, which we can see in the architecture of the system, through its use of small components that are fast to restart, along with cloud services, which have been built with resiliency as a primary characteristic.

## MTBF and MTTR versus RPO and RTO

A common error we see inexperienced architects make is to focus almost entirely on functional availability—as characterized by MTBF and MTTR—and to leave data availability as an afterthought. This approach is similar to how many systems are designed from a purely functional perspective with the design of the system’s data being something of an afterthought too.

Dealing with data in the face of failures is different from dealing with our functional components. If a functional component fails, we can just restart or replicate it; there is nothing to lose. In the case of data, however, we must avoid loss as well as maintain availability.

The key concepts we need to think about for the data availability in our system are the recovery point objective (RPO) and the recovery time objective (RTO). The RPO defines to how much data we are prepared to lose when a failure occurs in the system. The RTO defines how long we are prepared to wait for the recovery of our data after a failure occurs, which obviously contributes to the MTTR.

The two concepts are related: if we have an RPO of infinity (i.e., we are prepared to lose all our data if needed), we can clearly achieve an RTO of close to zero—although, of course, this is unlikely to ever be the case. If we need an RPO of zero, then we may have to accept a longer RTO (and perhaps other tradeoffs, such as reduced performance) due to the mechanisms, such as logging and replication, used to mitigate the risk of losing data.

Both measures apply at the system level and the individual component level. Considering both levels allows us to think about the importance of different sorts of data to the operation of our system and potentially prioritize different types differently.

Once again, we need to think clearly about our business context to establish what RPO and RTO levels we can live with so that we can design a system to meet these levels of resilience.

In the TFX context, all of the data related to payments is clearly critical and will need a very low RPO, aiming for no data loss. Therefore, we might need to accept a relatively long RTO for the Payment Gateway service, perhaps 20 or 30 minutes. Other parts of the system, such as the Fees and Commissions Manager’s data, could accept the loss of recent updates

(perhaps up to half a day's worth) without significant business impact, as re-creating the updates would not be particularly difficult. This means that we might be able to recover most of the system very quickly, probably within a minute, but with some services taking longer to recover and the system offering only partial service until they are available (e.g., offering partial service for a period after a major incident by recovering most of the system very quickly but not allowing payment dispatch or document updates for a longer period).

## Getting Better over Time

As noted, one of the competencies of a resilient organization is the ability to know what has happened in the past and to learn from it. This is an example of a feedback loop, of the sort we discussed in [Chapter 2, “Architecture in Practice: Essential Activities,”](#) and involves positive lessons as well as the more common lessons that organizations learn from mistakes. An open culture that stresses learning and sharing encourages feedback naturally and allows the successful practices that some parts of the organization discover to be adopted and tailored by other parts of the organization. In such a culture, hard-won lessons can be shared to avoid similar problems in the future. Sensible though this sounds, we have seen a couple of problems in achieving this ideal state in practice.

First, although people often say they want to share and learn from successes and failures, it sometimes just doesn't happen, perhaps because people don't feel safe in sharing failures or respected when they share their successes. Or perhaps it's because, despite best intentions, this sort of activity is never prioritized highly enough to allow people to focus on it and make it effective.

Second, when people engage in retrospective analysis, it must be based on an analysis of data collected at the time of the incident. A common problem with retrospectives without data is that they result in conclusions that are based on incomplete memories combined with opinions rather than an analysis of facts. This rarely leads to valuable insights!

Finally, we have often seen good retrospective analysis and sharing of successes and failures but with little long-term impact. This happens when there is no plan for implementing the insights and ideas from the exercise.

Implementing continual improvement takes time, resources, and energy. A resilient organization needs to invest all three in learning from its past to improve its resiliency in the future.

## The Resilient Organization

As engineers, we have a natural tendency to focus on technical solutions to complex problems, which often manifests in a lot of time spent on technical mechanisms for resilience but without much attention paid to other aspects of the problem. As we said at the start of this section, resiliency, like other quality attributes, involves the intersection of people, processes, and technology, and a resilient organization needs to work hard on all three, as principle 6, *Model the organization of your teams after the design of the system you are working on*, reminds us. There is little point in having sophisticated and resilient technology if we don't have operational processes to run it in a resilient manner; strong operational processes are unlikely to compensate for poorly trained or demotivated people to implement them.

In this book, we talk mainly about the design of the technology mechanisms to achieve resilience, but the resilient organization needs to ensure that the combination of its people, processes, and technology combine to create an entire environment that exhibits resiliency in the face of uncertainty. We provide some references in the “[Further Reading](#)” section that discuss the other aspects of resiliency in more depth.

## Architecting for Resilience

### Allowing for Failure

If we are to create a resilient system, we need to remember some of our Continuous Architecture principles. Principle 2, *Focus on quality attributes, not on functional requirements*, reminds us to think about possible faults and failures and how to mitigate them throughout the development life cycle and not hope that this quality can be added at the end. Like most qualities, resilience is best addressed by using principle 3, *Delay design decisions until they are absolutely necessary*, and work on resilience concerns little and often rather than at the end of the design process when a

lot of decisions have already been made. Principle 5, *Architect for build, test, deploy, and operate*, helps to ensure that we are thinking about how to integrate resilience testing into our work and the impact of resilience requirements on system operation.

Like other quality attributes, we have found a good way to explore the resilience requirements of a system is to use quality attribute scenarios—situations that the system may face and how it should respond in each one—described in terms of the stimulus, response, and measurement.

Once we have a representative set of scenarios for the resilience of our system, we can understand the real requirements in this area and explore our design options for meeting them.

If our requirements turn out to be more traditional high-availability requirements, where our system is deployed to a limited number of nodes that can be managed by high-availability clustering software and can absorb some downtime in case of severe failure, then a classical high-availability approach can be considered. This allows us to achieve the requirements using technologies such as clustering, whereby we push the availability mechanisms out of the application software into the infrastructure. If our system is running at a scale where high-availability technology won't work well or if we find that we cannot allow for recovery time, then we probably need to add resilience mechanisms in the design of all of our components. Or we may need a mix of high-availability and resilience mechanisms (e.g., using high-availability features in services such as databases and resilience mechanisms in our custom software). In either case, we need to make sure that we also address the people and process aspects of achieving system availability in the face of possible failure.

As an example, let us review some of the scenarios that the TFX team has identified to characterize the type of resilience needs of our example system, TFX, in operation.

<b>Scenario 1</b>	<b>Unpredictable performance problems in document store database</b>
<i>Stimulus</i>	Response times for the cloud platform document database service operations become unpredictable and slow, from normal operation of 5–10ms to 500–20,000ms.
<i>Response</i>	Initial performance degradation of requests involving stored documents is acceptable, but average request processing time should quickly return to normal, even if some functional limitations are required. During this scenario, the increased service response times and database service slowness should be visible to those operating the system.
<i>Measurement</i>	For up to 1 minute, requests to the Document Manager are slower than normal, taking up to 22,000 ms to complete, and unpredictable in duration. TFX clients and API users see unpredictable response times if document access is needed. After 1 minute, average request processing time should have returned to normal.
<b>Scenario 2</b>	<b>Very high API load condition</b>
<i>Stimulus</i>	Unprecedented peaks in request volume (15k/sec) are repeatedly experienced at the UI Service’s external API for periods of 30–60 seconds.

<i>Response</i>	When the level of requests being received is too large for the UI Service to deal with, requests that cannot be handled should be rejected with a meaningful error. The situation should be visible to those operating the system.
<i>Measurement</i>	Total number of requests received for the UI Service is abnormally large; response times for successful requests are largely unaffected (average time increased by less than 10%); the number of rejected requests from the UI Service is large (in the region of 30%).
<b>Scenario 3</b>	<b>Restart of single system service</b>
<i>Stimulus</i>	Fees and Commissions (F&C) Manager is deployed with incorrect configuration, requiring a redeploy, 1-minute shutdown for database change, and service restart.
<i>Response</i>	Requests that involve F&C Manager functions may slow down or fail for up to a minute, but minimum number possible should fail entirely. Unavailability of F&C Manager and slow and failed response levels should be visible to those operating the system.
<i>Measurement</i>	F&C Manager metrics and health check indicate unavailability for $\leq$ 1 minute, TFX requests needing F&C functions may fail or exhibit an average response time degradation of up to 200% of the normal average response time for $\leq$ 1 minute.
<b>Scenario 4</b>	<b>Unreliability of container service</b>
<i>Stimulus</i>	The cloud container service used to host our service containers starts behaving unpredictably: containers may not start, those running may not be able to invoke services on other containers, and some containers may be stopped unexpectedly.
<i>Response</i>	All parts of TFX may behave unpredictably, but the API for the mobile apps and possible external callers should remain available, even if they must return well-defined errors. The problems with service availability and container infrastructure should be visible to those operating the system.
<i>Measurement</i>	All metrics indicating service health should indicate problems, and health checks should indicate where problems occur.

When we review these scenarios, we can quickly see why they were identified. The first one indicates how TFX should respond to problems

with cloud resources it uses, the next indicates how it should respond to unexpected external events, the third defines how it should behave when its own software encounters a problem, and the fourth defines how it should behave if there is a significant problem with the cloud provider's platform.

When we review these scenarios, we can see a significant emphasis on uninterrupted service, even during significant failures of our system components and the underlying platform, although the level of service offered can be degraded to cope with the problems being suffered by the system. This emphasis points toward a resilience approach rather than a failover- and fallback-based high-availability approach.

To achieve technical resilience, we need to have mechanisms to

- *Recognize* the problem so that we can diagnose it and take suitable remedial action as quickly as possible to limit the damage and time taken to resolve the problem, using mechanisms such as *health checks* and *metric alerts*.
- *Isolate* the problem, to prevent an issue with one part of the system rapidly spreading to become a problem with the whole system, using mechanisms such as *logical or physical separation* in our deployment environment, or *failing fast* when making requests, to prevent problems in one component propagating through to a component trying to use it.
- *Protect* system components that become overloaded. To avoid overloading parts of the system that are already under heavy load, use mechanisms such as *controlled retries*, which limit the rate at which failed operations are retried, to reduce the workload of a component with a problem.
- *Mitigate* the problem, once recognized, ideally using automated means but involving manual procedures if needed, through mechanisms such as *state replication*, to allow recovery or failover with minimal data loss, and *fault-tolerant request handling*, to allow for transient problems through use of fault-tolerant connectors such as queues rather than remote procedure calls (RPCs).
- *Resolve* the problem by confirming that the mitigation is effective and complete or identifying further intervention that is required (possibly

through manual means) using mechanisms such as *controlled restarts* of system components or *failover* from one resilience zone<sup>5</sup> to another.

<sup>5</sup> Such as an Amazon Web Services or Azure Availability Zone.

The TFX team can work through the scenarios, considering which architectural tactics to apply to achieve each of these resilience properties to address the challenges of that specific scenarios (and implicitly other similar ones). We discuss some of the specific architectural tactics that are likely to be useful in the next section of this chapter.

As well as the architecture of the software and mechanisms within it, the team also needs to ensure that they have a strategy for recovery of the entire system (for situations such as the unreliability of container service scenario, major cloud platform performance or scalability problems, or complete failure of an environment, such as a data center or a public cloud resilience zone). As we discuss in this chapter, the system also needs monitoring mechanisms in place, such as tracing, logging, metrics, and health checks, to provide visibility of what is going on within TFX and allow rapid problem identification.

We explore how to achieve one of the example resilience scenarios under “[Achieving Resilience for TFX](#).”

## Measurement and Learning

As architects, our natural focus is on solving system-level design problems and working out the best way to build the software. However, as we’ve discussed in this and earlier chapters on performance, scalability, and security, to achieve these system qualities, we also need to be able to measure the behavior of our system in operation and use the information we gain to learn how to improve its performance, scalability, security, and resiliency in the future. Doing so involves understanding what is going on today, knowing what has happened in the past, and projecting what could happen in the future.

While often seen as an operational concern, it is important to start measuring and learning as early as possible in the system delivery life cycle to ensure that we maximize resiliency from the beginning and embed measurement and learning in the technology and culture of our system.

Despite starting early, however, we should also remember principle 3, *Delay design decisions until they are absolutely necessary*, to avoid wasted work. The way to do this is to start measuring what we have and learning from it, extending the mechanisms as we need them, rather than building a comprehensive and complex measurement infrastructure at the start. As long as we know how to extend our approach as our system scales up, starting small is a good strategy.

An obvious point that is sometimes missed is that learning from success is as important as learning from failures. From an incident retrospective, we can learn about mistakes we made and how to avoid them, and people intuitively do so. However, there is also a lot to learn from a successful period, perhaps when we released many changes without any problems or dealt with an unexpected peak load condition. Why was it that we were resilient in these situations? Did people anticipate and avoid problems? Did automated mechanisms encounter predictable situations and work well? Did we have good processes that avoided human error? Was it just luck in some cases? By understanding what we have done well and done badly, we generate a stream of learning opportunities to improve our resilience.

The concrete activities that we need to achieve measurement and learning are as follows:

- *Embed measurement mechanisms* in the system from the beginning so that they are standard parts of the implementation and data is collected from all the system's components.
- *Analyze data regularly*, ideally automatically, to understand what the measurements are telling us about the current state of the system and its historical performance.
- *Perform retrospectives* on both good and bad periods of system operation to generate insight and learning opportunities for improvement.
- *Identify learning opportunities* from data and retrospectives, and for each, decide what tangible changes are required to learn from it.
- *Improve continuously and intentionally* by building a culture that wants to improve through learning and by prioritizing improvement (over

functional changes, if needed) to implement the changes required to achieve this.

We discuss some of the tactics and approaches to achieve measurement and learning in the following sections of this chapter and provide some useful sources of information in the “[Further Reading](#)” section.

## Architectural Tactics for Resilience

It isn’t possible for us to give you a single architectural pattern or recipe to make your system resilient. Creating a resilient system involves many architectural decisions at different levels of abstraction that, when combined, allow the system to survive failures of different types.

Nevertheless, we have found some architectural tactics and decisions to be generally valuable for increasing resilience, and we discuss them in this section. We don’t claim that this list is comprehensive, but it is the set of tactics and decisions that we have found to be generally useful.

The tactics are grouped according to which element of resilience they primarily address: recognizing failures, isolating failures, protecting system components to prevent failures and allow recovery, and mitigating failures if they do occur. We don’t have tactics for the resolution of failures, as failure resolution is generally an operational and organizational element of resilience that isn’t directly addressed by architectural tactics. It is also perhaps obvious, but worth noting, that for the four elements of resilience where we do have tactics, using these tactics is only part of achieving that element of resilience. As well as applying the tactics intelligently, we need to make sure we implement the tactics correctly, test resilience thoroughly, and have the operational mechanisms and procedures to react to incidents during system operation.

### Fault Recognition Tactics

#### *Health Checks*

One of the most basic things we can do to check that the components of our system are working correctly is to ask them. This means making an architectural decision to implement a standardized health check function in

every component of the system that typically runs some dummy transaction through the component to check it is working correctly.

The most common approach to implementing health checks is to add an API operation to every component (ideally with a standard name and calling conventions) that instructs the component to perform a self-check of its health and return a value indicating whether all is well or not. If the health check operation can't be called or doesn't return, that obviously indicates a problem too.

The difficulty with implementing health checks is knowing what to do to verify that the service is working correctly, as there are many things that you could check. Our view is that the most effective way to perform the health check is to perform “synthetic” transactions that are as close as possible to the real transactions that the service performs but do not affect the real workload of the system. For example, in our TFX case study system, the Fees and Commissions Manager health check could call the internal operations to calculate a representative fee, perhaps for an imaginary test client, and perform an update to the configuration for that client (thereby preventing any unintended side effects on normal workload). This checks that the service can read data, write data, and perform its key functional responsibilities. Clearly, care must be taken not to do anything in the health check that can cause a problem for the real workload of the system (such as writing something to a database that could cause incorrect results to be returned for a real API call).

Once you have health checks in all of your components, you need a way of using them. They are often required for load balancing, as load balancers need some way of establishing that a service instance is healthy, and when used in this way, they need to respect the specific requirements of the specific load-balancing software in use. Health checks can also be called from system-specific monitoring components that allow a visual dashboard of some sort to be created for the system's operators and allow alerts to be raised if health checks indicate problems.

## ***Watchdogs and Alerts***

Following on logically from health checks are the tactics of using watchdogs and raising alerts. A *watchdog* is a piece of software whose only

responsibility is to watch for a specific condition and then perform an action, usually creating some form of alert, in response.

Many cloud and on-premises monitoring tools provide comprehensive watchdog facilities available via simple configuration, usually with the ability to call an operating system script for more complex or context-specific checks. Generally, using a well-tested monitoring platform to create watchdogs is a good decision, although it does tie us to the tool or platform.

It is of course possible, and fairly common, to have custom watchdog scripts running in the operational environment, perhaps providing some sort of monitoring that the platform can't do or implemented simply because it was the quickest solution when the watchdog was needed. We caution against a lot of ad hoc watchdog processes, though (or, indeed, many ad hoc operational mechanisms at all), as they can quickly become a zoo of poorly understood scripts that do little to aid system resilience. This is a case when remembering that principle 5, *Architect for build, test, deploy, and operate*, is particularly relevant, making sure that a systemwide approach is taken to fault recognition.

Watchdog processes normally create some sort of alert in response to a condition being recognized. Alert handling varies from simple emails and text messages sent directly by the script to sophisticated, dedicated systems<sup>6</sup> that provide full **incident management**, including alerting, escalation, handling of on-call rotas, prioritization, stakeholder communication, and incident life cycle management, all integrated with many other widely used tools. How alerts are handled is probably more of an operational decision than an architectural decision, but once made, the important thing is that alerts are consistently created and handled so that they are a reliable warning mechanism for your system, which is an architectural concern.

<sup>6</sup> Such as PagerDuty or Opsgenie, to name two popular examples at the time of writing.

In the TFX system, our team would probably use the monitoring systems provided by the cloud platform they are using, with extensive configuration to monitor all of the components of the system and watch for TFX-specific conditions that indicate potential problems. Initially, they might well use a simple approach to handling alerts, perhaps just sending messages and raising tickets in their issue-tracking system, moving later to a full incident

management system as the complexity of their operational environment increases.

## Isolation Tactics

### ***Synchronous versus Asynchronous Communication: RPCs versus Messages***

When most people think of service-based systems or follow a tutorial on building microservices using their favorite framework, the communication between services (the **connector** between the components in architecture jargon) is nearly always assumed to be simple synchronous JavaScript Object Notation (JSON) over HTTP. Simple, quick, reasonably efficient, and interoperable—what is not to like?

I’m sure you know where we are going with this. Simple HTTP-based RPCs using GET and POST requests are straightforward and performant, but they are also quite fragile because they are *synchronous* connectors. If something goes wrong with the network or with a key service, it is easy to end up in a situation with half your system blocked waiting for responses to arrive. This means that faults propagate quickly through the system to become visible failures and can become difficult to recover from. From the service’s perspective, synchronous inbound requests can also be a problem because the service can’t control when they arrive, and they need to be handled immediately. Consequently, a huge number can arrive at once, causing a massive amount of parallel request handling, which can overwhelm the service. It is certainly possible to mitigate both the client and server ends of this problem, but it adds quite a lot of complexity to both (which can have its own impact on reliability).

As discussed in [Chapter 5, “Scalability as an Architectural Concern”](#), the alternative is to use some form of inherently *asynchronous* connector, the classical one being point-to-point message queues, which allows asynchronous client behavior whereby clients enqueue requests and then listen on another queue for a response. A common variation on point-to-point message queues is publish/subscribe messaging, which allows sharing of the message channels. In both cases, the messaging system provides buffering of messages between sender and receiver. You can, of course, also

create asynchronous connections using other transports, such as HTTP, although this approach is more complex for the client and server and is less common.

Message-based asynchronous communication allows both client and server to manage faults and high load conditions much more easily than if they are linked using synchronous connectors. If a server needs to be restarted, requests can still be sent during the restart and processed when the server is available again. In a high load situation, the server continues to process requests, and messages build up in the message queue or bus if they are arriving faster than they are processed (possibly eventually filling the queue and returning an error to the clients). As we saw in Chapter 5, this approach also provides a straightforward way to scale the server by having many instances read the message queue or bus to handle the incoming requests. The tradeoff is a more complex client that must handle results arriving independently of the request (perhaps in a different order) and the additional environmental complexity of the messaging middleware (which itself must be resilient!). As mentioned in [Chapter 6, “Performance as an Architectural Concern,”](#) there is also usually a performance-related tradeoff, as message-based communication is usually higher latency and less efficient than RPC-based communication because of the overhead of the messaging middleware between the client and server. In our experience, these tradeoffs are well worth accepting in situations where resilience is important. It is significantly easier to make an asynchronous system resilient compared to a highly synchronous one.

In our TFX case study, the team could apply this tactic between the UI Service and the other domain services (e.g., the Document Manager and Fees and Commissions Manager). This would improve the resilience of the internal system communication while retaining the ease of use and low latency of RPC requests for the user interfaces. As you will remember from the [Chapter 5](#), the team also concluded that they might need to use asynchronous communication in places for scalability reasons, and they decided to use a standard library for intercomponent communication to encapsulate the details of this from the main application code, so this tactic benefits both quality attributes.

## **Bulkheads**

One of the hallmarks of a resilient system is its ability to gracefully degrade its overall service level when unexpected failures occur within it or unexpected events occur in its environment. An important tactic for achieving graceful degradation is to limit the scope of a failure and so limit the amount of the system that is affected (or “limit the blast radius,” as people sometimes say, rather dramatically).

Bulkheads are a conceptual idea; they can contribute significantly to limiting the damage from an unexpected fault or event. The idea of a bulkhead, borrowed from mechanical engineering environments such as ships, is to contain the impact of a fault to a specific part of the system to prevent it becoming a widespread failure of service.

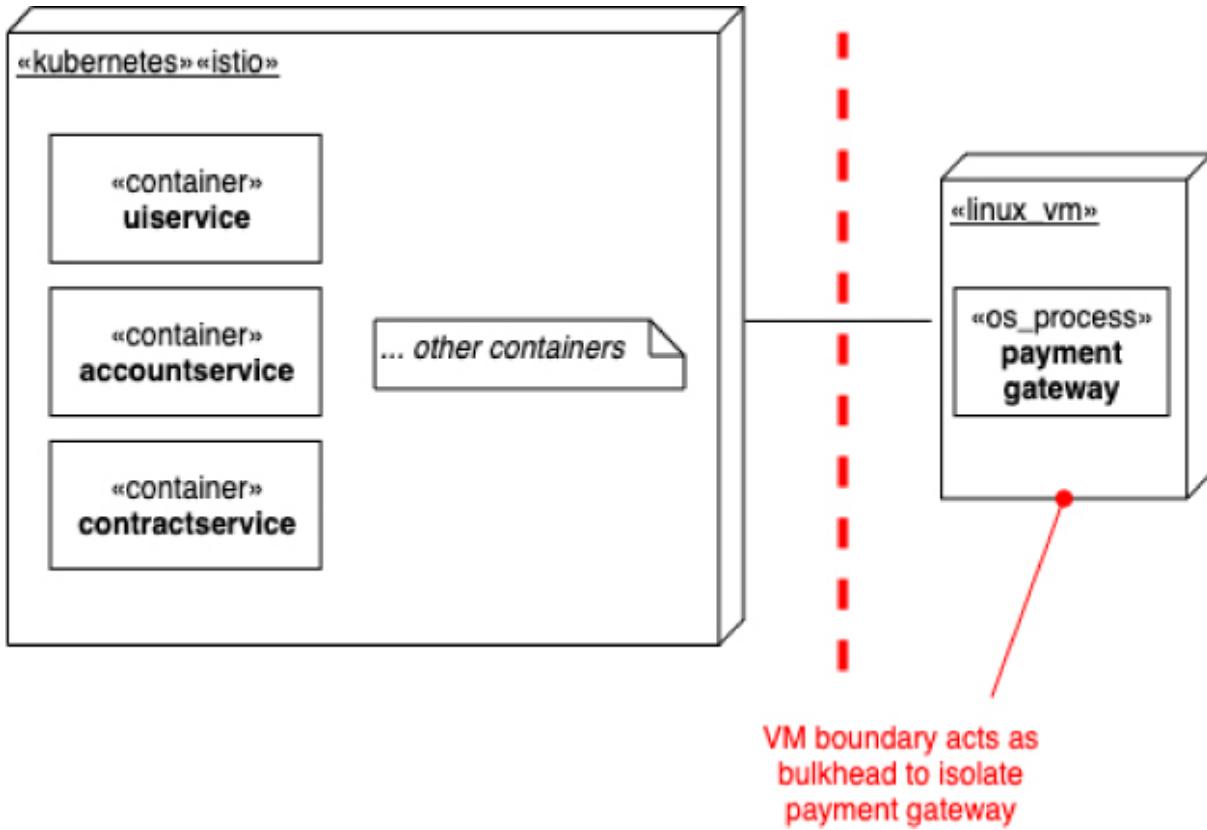
Obviously, in our software-based world, we cannot install physical walls to limit physical effects, but there are quite a few mechanisms that can be used to achieve a bulkhead-like isolation between parts of our system.

Redundancy and separation of workload in our deployment environment, such as separate servers for different parts of the system, or using execution policies in a **Kubernetes** service mesh to force separation of different groups of services are two examples. At a finer-grained level, we can manage the threads in a process so that we dedicate different threads to different workloads, ensuring that problems with one type of workload do not cause the entire process to stall.

Although more obvious in a physical computing environment, even in a cloud environment where everything is virtual, we can choose the degree of separation between our system components, from rudimentary separation of services into different virtual machines at one extreme to separate deployment environments in different parts of the cloud platform (e.g., regions, or **availability zones**) at the other, depending on the degree of isolation required.

Whichever way we choose to create our separation, the bulkhead is a useful conceptual device to force us to consider which parts of the system are most important and which parts can operate usefully independently. Once we understand which parts of our system can operate independently, we can identify where we need to limit fault propagation and choose appropriate mechanisms to achieve it.

In the TFX system, the team have to use a third-party payment gateway, which might have unpredictable resource demands that could cause problems if running in a shared execution environment. As shown in [Figure 7.2](#), a resource-limited virtual machine (VM) could act as a bulkhead to isolate it.



**Figure 7.2** A virtual machine acting as a bulkhead

### **Defaults and Caches**

Whenever we are considering the resilience of our system, we need to keep asking, What will happen if that service is unavailable for a period?

Defaults and caches are mechanisms that can help us provide resiliency in some situations when services that provide us with data suffer failures.

As discussed in the [Chapter 5](#), a cache is a copy of data that allows us to reference the previous result of a query for information without going back to the original data source. Naturally, caches are suitable for relatively slow-changing information; something that varies significantly between every call to the service is not a useful data value to cache.

Caches can be implemented using any data storage mechanism, from simple, in-memory data structures private to each process (or even thread) in the system to elaborate network-accessible stores such as Redis, which is very useful when retrieving or storing the cached value incurs significant cost, making it sensible to share the value between callers that want to use it. Caches can also be *inline*, implemented as proxies (and so the caller is unaware of the cache) or as *lookaside* caches where the caller explicitly checks and updates the cache value.

An example in the TFX system could be a cache built into the Fees and Commissions Manager to cache the results of complex lookups and calculations to calculate the fees and commissions for different types of transaction. Once the results are returned to the caller, intermediate results or lookup results could be stored in a memory cache to be reused. Cached fee and commission definitions can be invalidated by the Fees and Commissions Manager when they change, as it manages this data. Data that it looks up could be invalidated after a time period or as a result of an operational action if the cache was known to be stale. For example, as we mentioned in [Chapter 3, “Data Architecture,”](#) the Fees and Commissions Manager needs to look up details of the letter of credit (L/C) counterparties, and these would be good candidates for caching.

Of course, there is always a tradeoff involved in any architectural mechanism and, as the old joke goes, there are only two hard things in computer science: cache invalidation, naming things, and off-by-one errors. The big problem with caches is knowing when to discard the cached value and call the service again to refresh it. If you do this too often, you lose the efficiency value of the cache; if you do it too infrequently, you are using unnecessarily stale data, which may be inaccurate. There isn’t really a single answer to this problem, and it depends on how often the data is likely to change in the service and how sensitive the caller is to the change in value.

From a resilience perspective, a cache can be combined with a fast timeout to provide a resilient alternative when the service fails. If a caller finds that its call to the service times out, it just uses the value in the cache (as long as there is one). If the call succeeds, the caller can update the value in the cache for use later if needed.

Finally, default values can be viewed as a primitive kind of cache with predetermined values configured or coded into the caller. An example of where default values work well is systems with user personalization features. If the service to provide the personalization parameters is not available, a set of default values can be used to provide results rather than have the request fail. A set of default values is much simpler to use than a cache is correspondingly more limited and may well be acceptable in only relatively few cases.

## Protection Tactics

### ***Back Pressure***

Suppose we do have an asynchronous system but one where the clients sometimes produce workload a lot faster than the servers can process it. What happens? We get very long request queues, and at some point, the queues will be full. Then our clients will probably block or get unexpected errors, which is going to cause problems.

A useful resilience tactic in an asynchronous system is to allow some sort of signaling back through the system so that the clients can tell that the servers are overloaded and there is no point in sending more requests yet. This is known as creating *backpressure*.

There are various ways you can create backpressure for the caller, including making the write to the queue a blocking write when the queue is full, returning a retry error when the queue is full, or using features of the middleware provider to check for queue status before sending the request. Backpressure mechanisms like these allow you to deal with high load conditions that could otherwise cause complex system failures as timeouts and blocked requests cascade back up the request tree, causing unpredictable results.

The limitation of using backpressure is that it relies on the client behaving correctly. Backpressure is just a signal to the client to reduce its processing speed to reduce load on the server. There is nothing the server can do if a poorly implemented or malicious client ignores the backpressure signal. For this reason, backpressure needs to be combined with other tactics, such as load shedding, which we discuss next.

In our case study, if the TFX team decided to implement message-based communication from the UI Service to the other services, as suggested as an option earlier, then they could fairly easily indicate to the UI Service whether the other services were overloaded, allowing a clear return code indicating so to the user interfaces.

## ***Load Shedding***

Just as the easiest way to achieve better performance in a system is to identify a design that requires less work to achieve the required output, a valuable architectural tactic for resilience is to simply reject workload that can't be processed or that would cause the system to become unstable. This is commonly termed *load shedding*.

Load shedding can be used at the edge of the system, to reject work as it enters the system, or within the system to limit work in progress. It is particularly useful at the system boundary, where the use of backpressure, which would probably block the caller (as discussed previously), might cause as many problems as it solves. Load shedding rids the system of the work entirely, by returning a clear status code indicating that the request cannot be handled and should not be retried immediately (such as an HTTP 503, service unavailable, or HTTP 429, too many requests). With a sensibly written requesting client, this can provide enough of a gap in request traffic to allow the system to process its backlog or for human intervention if required. However, even if the client ignores the error code and retries immediately, the request will still be rejected (unlike with backpressure).

Another attraction of load shedding is that it can normally be implemented relatively easily, using standard features of infrastructure software such as API gateways, load balancers, and proxies, which can normally be configured to interpret an HTTP return code as a signal not to use a service instance for some period. For example, in the TFX system, the team is using an API gateway, as we see later in this chapter.

A variant of load shedding that is sometimes identified as a distinct approach is *rate limiting*<sup>7</sup> (or “throttling”). Rate limiting is also normally provided by infrastructure software that handles network requests. The difference between the two is that load shedding rejects inbound requests because of the state of the system (e.g., request handling times increasing or

total inbound traffic levels), whereas rate limiting is usually defined in terms of the rate of requests arriving from a particular source (e.g., a client ID, a user, or an IP address) in a time period. As with load shedding, once the limit is exceeded, additional requests are rejected with a suitable error code.

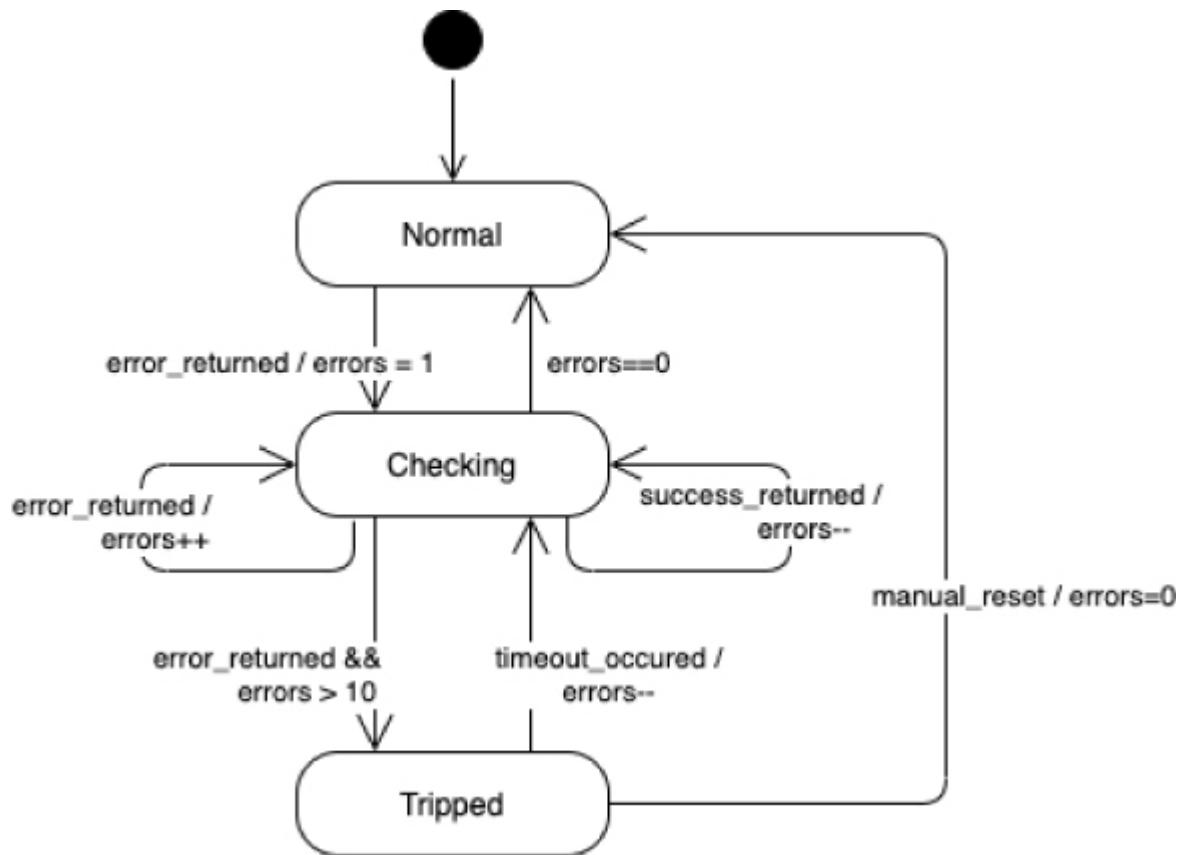
<sup>7</sup> It would be fair to say that none of these terms is universally agreed, and they evolve as people innovate and create new solutions, but Daniel Bryant from the Ambassador Labs team has written a clear explanation of how they define rate limiting and load shedding: “Part 2: Rate Limiting for API Gateways” [blog post] (May 8, 2018). <https://blog.getambassador.io/rate-limiting-for-api-gateways-892310a2da02>

## ***Timeouts and Circuit Breakers***

While bulkheads can be used to minimize the impact of failure within a group of system services, timeouts and circuit breakers are used to minimize the impact of service problems on both the caller and the service itself.

Timeouts are a very familiar concept, and we just need to remind ourselves that when making requests to a service, whether synchronously or asynchronously, we don’t want to wait forever for a response. A timeout defines how long the caller will wait, and we need a mechanism to interrupt or notify the caller that the timeout has occurred so that it can abandon the request and perform whatever logic is needed to clean things up. We mentioned the principle of failing fast in [Chapter 5](#), and this is a good principle to bear in mind when designing for resilience. We don’t want a lot of outstanding requests waiting for responses that are unlikely to arrive. This can quickly tip us into an overload condition if load is high. So, while there is no magic formula for choosing the correct timeout, we need to make them relatively short, perhaps starting with 150 percent of the average expected service response time and tuning from there.

A timeout helps to protect the client from being blocked if a service fails or has encountered a problem, whereas a circuit breaker is a mechanism at the requestor end aimed primarily at protecting the service from overload. A circuit breaker is a small, **state machine**–based proxy that sits in front of our service request code. An example statechart for a circuit breaker is shown in [Figure 7.3](#).



**Figure 7.3** Circuit breaker statechart

The statechart uses Unified Modeling Language (UML) notation, with the rounded rectangles representing the possible states of the circuit breaker, the arrows indicating possible state transitions, and the text on the transitions indicating the condition, before the slash (/), that causes the transition to occur and the text after the slash indicating any action that occurs during the transition.

There are many ways to design a circuit breaker proxy to protect a service from overload, but this example uses three states: normal, checking, and tripped.

The proxy starts in normal state and passes all requests through to the service. Then, if the service returns an error, the proxy moves into checking state. In this state, the proxy continues to call the service, but it keeps a tally of how many errors it is encountering, balanced by the number of successful calls it makes, in the errors statechart variable. If the number of errors returns to zero, the proxy returns to normal state.

If the errors variable increases beyond a certain level (10 in our example) then the proxy moves into tripped state and starts rejecting calls to the service, giving the service time to recover and preventing potentially useless requests piling up, which are likely to make matters worse. Once a timeout expires, the proxy moves back into checking state to again check whether the service is working. If it is not, the proxy moves back into tripped state; if the service has returned to normal, the errors level will rapidly reduce, and the proxy moves back into normal state.

Finally, there is a manual reset option that moves the proxy from the tripped state straight back to normal state if required.

Both timeouts and circuit breakers are best implemented as reusable library code, which can be developed and tested thoroughly before being used throughout the system. This helps to reduce the likelihood of the embarrassing situation of a bug in the resilience code bringing down the system someday.

In TFX, the team could build little libraries for user interfaces that “decorate” standard operating system libraries for making API requests to make them easier to use for TFX’s services and build in a circuit breaker mechanism to prevent the user interfaces inadvertently overloading application services, without complicating the main code of the user interface components.

## Mitigation Tactics

### ***Data Consistency: Rollback and Compensation***

When things go wrong in a computing system, one of the significant complications is how to ensure that the state of the system is consistent in the case of faults. The recognition of this need decades ago led to the development of a whole subfield of computer science: transaction processing. This area of study has enabled reliable relational databases, **ACID transactions**, reliable processing of billions of bank and payment transactions per year, and many other modern miracles.

We talk more about data consistency and the tradeoffs needed to achieve it in [Chapter 3](#), but simple transactions with a single database involve marking the beginning of the transaction (which is often implicit),

performing one or more data update operations, and then marking the end of the transaction as completed, either as “committed” (i.e., success—save the changes) or “rollback” (i.e., failure—discard the changes). The complexity is hidden within the database, as it guarantees that the transaction will be either entirely completed or entirely discarded, even in the case of client or database failures. Such transactions are known as ACID transactions<sup>8</sup> because they are atomic, consistent, isolated, and durable, and they provide *strong consistency* where (simplistically) the developer can ignore concurrency and data replication or distribution due to the transactional guarantees.

<sup>8</sup> Wikipedia, “ACID.” <https://en.wikipedia.org/wiki/ACID>

Things get complex when we have multiple databases (or “resource managers,” in the jargon of transaction processing),<sup>9</sup> and so we have to coordinate the transactions in each database to form a single master (or distributed) transaction. The most common way to achieve this is the famous (or infamous) two-phase commit (2PC) algorithm.<sup>10</sup>

<sup>9</sup> Resource managers are described at Wikipedia, “X/Open XA.” [https://en.wikipedia.org/wiki/X\\_Open\\_XA](https://en.wikipedia.org/wiki/X_Open_XA). The technical standard is available from The Open Group, “Distributed Transaction Processing: The XA Specification” (1991). <http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf>

<sup>10</sup> Wikipedia, “Two-Phase Commit Protocol.” [https://en.wikipedia.org/wiki/Two-phase\\_commit\\_protocol](https://en.wikipedia.org/wiki/Two-phase_commit_protocol)

2PC is a sophisticated and well-validated approach to achieving transactional consistency across multiple databases. However, in use, it has proved to be operationally complex, exhibit poor performance, and difficult to make reliable in practice.

This situation means that if we have a system with more than one database (which is more and more common today) and we have requests to our system that may involve updating more than one of the databases, we have a potential consistency problem if we have faults in the system.

Our options are to limit our requests to only ever updating a single database, to use a transaction processing monitor to implement distributed transactions, or to have some way of ensuring sufficient consistency without distributed transactions.

In many modern systems, with multiple databases, we simply can't limit requests to updating a single database without making our API impossibly difficult to use (and just shifting the consistency problem to our callers). Historical experience suggests that the use of 2PC protocols does not work very well at scale.<sup>11</sup> This leaves us with the need to find another mechanism to ensure consistency when faults occur.

<sup>11</sup> Pat Helland, "Life beyond Distributed Transactions: An Apostate's Opinion" in *Proceedings of the 3rd Biennial Conference on Innovative DataSystems Research (CIDR)*, 2007, pp. 132–141.

The most common architectural tactic used to solve this problem is *compensation*, which is how we deal with potentially inconsistent data without an automatic transactional rollback mechanism being available. It is important to be aware that compensation does not guarantee strong consistency<sup>12</sup> as ACID transactions do but instead provides a form of weak consistency, where at the end of the compensation process, the different databases will be consistent. However, during the process, they will be inconsistent, and unlucky readers of the affected data will see inconsistent results (so-called phantom reads).

<sup>12</sup> Consistency is quite a complex topic, and we only scratch the surface of it here. Martin Kleppmann, *Designing Data-Intensive Applications* (O'Reilly Media, 2017) is an example of a text that provides a lot more depth on this subject.

You may remember that we discussed in [Chapter 3](#) another related tactic known as *eventual consistency*. The difference between the two is that compensation is a replacement for a distributed transaction and so actively tries to minimize the window of inconsistency using a specific mechanism: compensation transactions. Eventual consistency mechanisms vary, but in general, the approach accepts that the inconsistency exists and lives with it, allowing it to be rectified when convenient. Eventual consistency is also more commonly used to synchronize replicas of the same data, whereas compensation is usually used to achieve a consistent update of a set of related but different data items that need to be changed together.

Compensation sounds like a straightforward idea, but it can be difficult to implement in complex situations. The idea is simply that for any change to a database (a transaction), the caller has the ability to make another change that undoes the change (a compensating transaction). Now, if we need to update three databases, we go ahead and update the first two. If the update

to the third fails (perhaps the service is unavailable or times out), then we apply our compensating transactions to the first two services.

A pattern for organizing distributed transactions, which has been implemented in a number<sup>13</sup> of open source libraries,<sup>14</sup> is the Distributed Saga pattern, popularized by a number of well-known microservices consultants in recent years<sup>15</sup> but in fact dating back to original research from the database community in 1987.<sup>16</sup> A *saga* is a collection of compensating transactions, which organizes the data needed to perform the compensation in each database and simplifies the process of applying them when needed. When implemented by a reusable library, they can provide an option for implementing compensating transactions at a relatively manageable level of additional complexity.

<sup>13</sup> <https://github.com/statisticsnorway/distributed-saga>

<sup>14</sup> <https://github.com/eventuate-tram/eventuate-tram-sagas>

<sup>15</sup> For example, <https://microservices.io/patterns/data/saga.html>

<sup>16</sup> Hector Garcia-Molina and Kenneth Salem, “Sagas,” *ACM Sigmod Record* 16, no. 3 (1987): 249–259.

In our case study, the TFX team deliberately structured their services around domain concepts to avoid as many situations as possible where some form of distributed transaction would be required. They also chose to use relational databases, with conventional ACID transactions, for the majority of their databases. However, business requirements may well emerge that need an atomic update to two or more services, such as the Contract Manager and the Fees and Commissions Manager. In this case, the UI Service will probably need to include logic to perform this update safely, probably using a saga to do so in a structured way.

## ***Data Availability: Replication, Checks, and Backups***

In Chapter 3, we touched on the explosion in database models that has happened in the last few years with the growing popularity of NoSQL databases. Important factors in our resiliency design are the characteristics of the underlying database, the number of databases that we are using, and

whether we are using an RDBMS with ACID transactions or a partitioned NoSQL database with data replication and eventual consistency.

In that chapter, we reminded you of Brewer's CAP theorem, which neatly captures the relationship between consistency, availability, and partition tolerance (a type of resilience) in the context of a distributed data system. This reminds us that for a distributed data system, we must accept some tradeoffs, specifically that to maintain consistency (which is important to many systems), we will have to trade off partition tolerance or availability.

As noted earlier, data needs special attention when considering resilience, as the problem involves avoiding loss as well as maintaining availability. Depending on the level of failure we are concerned about, different architectural tactics can help mitigate a failure.

*Replication* can be used to mitigate node failure by ensuring that the data from the failed node is immediately available on other nodes (and as we discussed in [Chapter 5](#), data replication can be a useful scalability tactic too). Many NoSQL database products implement some version of this tactic as a standard feature, and some, such as Cassandra, use it as a fundamental part of their design. Most relational databases offer replication as an option.

If we suffer the failure of an entire service, then we can compensate for it in the short term using the caching and defaults tactic we outlined earlier, and to make the service data available again, restart the service in place, replicate the data to another instance of the service so it can be served from there, or start up an entirely new service (perhaps in another cloud resilience zone) to replace the failed service (which obviously relies on having replicated the data between the two—the latency of that replication defining the RPO that we can achieve).

If we lose an entire site (an entire cloud resilience zone or a traditional data center), then we need to be able to access our data from a surviving part of the cloud provider or the enterprise network. This nearly always involves setting up data replication between databases in the two zones and using a failover model to replace the services in the failed zone with new but equivalent services in the surviving zone. As with service failure, the latency in our data replication mechanism defines how much data we are likely to lose in this scenario and so defines the RPO that we can achieve.

Finally, there is the question of resilience in the face of the difficult problem of data corruption within our databases. Data corruption can be a result of defects in our system's software or problems or defects in the underlying database technology we are using. Such problems are thankfully relatively rare, but they do happen, so we need to have a strategy for them. The three tactics to use to mitigate data corruption are regular checking of database integrity, regular backup of the databases, and fixing corruption in place.

*Checking* a database involves both checking the underlying storage mechanism's integrity and checking the integrity of the system's data. Most database systems come with a mechanism for performing an internal consistency check (such as **DBCC** in SQL Server), and it is not terribly difficult to write scripts that can check the data model integrity of our system's data. The sooner we find a problem, the easier it will be to fix, possibly with no impact on operation if we find it early enough.

*Backup* of databases is an obvious strategy, although there is a subtlety to it. We obviously want to avoid backing up the corruption if possible. Therefore, it is worth understanding how the backup mechanism you are using works. Ideally, it will perform consistency checking itself, or it will unload the data into a format that is different from the physical structure of the database, bypassing internal corruptions in many cases (e.g., unloading RDBMS tables as rows rather than pages).

Finally, if we do find corruption, a strategy for dealing with it is to fix it in place by using database-specific tools for internal corruption (perhaps copying a database table's content to a new table, using a specific index to avoid internal corruption) or applying system specific "data fixes" to resolve system-level corruption once a defect is identified and fixed.

In our TFX system, the team has decided to use cloud services for their databases, so they are relying on the cloud provider to keep this software up to date and to run routine data integrity checking for them. Beyond this, however, they should be thinking about how to check the application-level integrity of their data, particularly across databases, perhaps building a routine reconciliation utility that can be run frequently. They will also need a strategy to deal with cloud availability zone problems (allowing failover to a surviving zone) and an effective data backup strategy.

## Achieving Resilience for TFX

We don't have space to consider all the work we would need to do to ensure the resilience of TFX, our case study system, but we see how the TFX team might work through one of the resilience scenarios to illustrate the process and show how they would use the information in this chapter to achieve resilience in that situation.

The scenario we work through is the very high API load condition scenario, where TFX suddenly experiences a very high request load on the UI Service API. The scenario makes it clear that the system should remain available under this load but can reject requests that cannot be processed, and the situation should be visible to those operating the system.

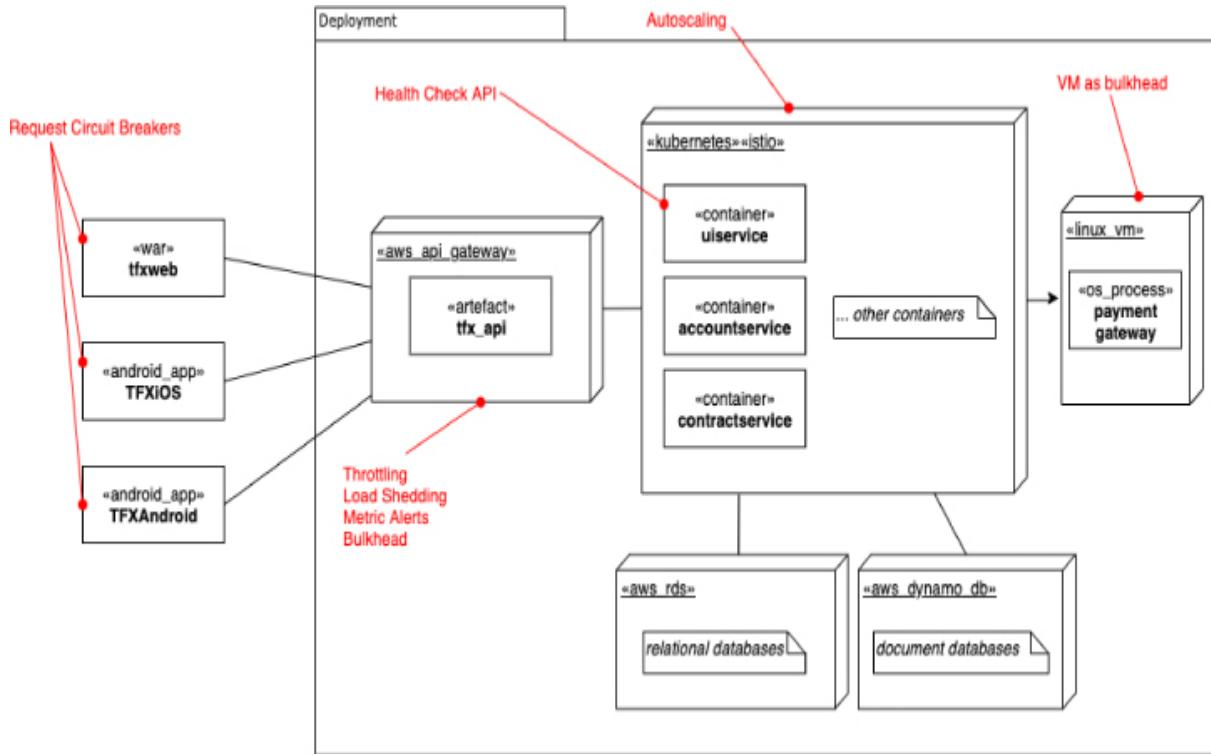
Examples of how the TFX team might meet the five elements of resilience for their system is summarized in [Table 7.1](#).

**Table 7.1** Resilience Activities for API Overload Scenario

Element	Mechanisms
Recognize	We will recognize this condition by implementing <i>health check</i> requests on the UI Service and, using an API gateway, monitoring alert for the API request rate. These will need to be integrated into an overall system monitoring dashboard.
Isolate	We will use a <i>bulkhead</i> to isolate the Payment Gateway from the rest of the system. The Payment Gateway is a third-party software component and historically has caused problems by consuming significant amounts of memory and CPU when under load. We will run it in a specifically sized Linux VM that acts as a bulkhead to prevent this piece of software affecting others should it act in this manner again.
Protect	We will build <i>circuit breaker</i> logic into our API requests from our user interfaces (the mobile apps and Web UI) to achieve controlled retries. This will prevent further overloading of the network service but will allow the user interfaces to continue working if the overload is transitory.
	The API gateway also provides protection, as it will use <i>load shedding</i> if request traffic levels become too high for us to handle.
Mitigate	To provide mitigation of this situation, we will configure our cloud platform to autoscale the services after 60 seconds of this condition. The autoscaling process is likely to take several minutes, so this mitigation should be effective in less than 5 minutes. However, we will build a human confirmation step into the automation so that an attack can be differentiated from high demand. If the cause is an attack, we would not autoscale to meet it, as it is not useful traffic, and we would engage a protection service such as Akamai, Arbor Networks, or Cloudflare.
Resolve	Resolving this resilience scenario does not require architectural design but does require us to have an operational strategy. If the request load subsides quickly, we will analyze it to consider whether we should have more online capacity in our system. If it does not subside quickly and is judged to be valid request traffic, we will confirm that autoscaling is effective. If the traffic appears to be an attack, we will try to identify characteristics that allow it to be discarded by the API gateway involving a third-party

... or even by another company, involving a third party specialist security company if required.

The architectural impact of meeting the requirements for this resilience scenario are highlighted in [Figure 7.4](#), which is a slightly altered version of the deployment view of our TFX case study (see [Appendix A](#)).



**Figure 7.4** Resilience tactics in the TFX deployment view

The notation for this diagram is loosely based on the UML deployment diagram. The cube shapes are execution environments, somewhere our system's components can be executed, such as a Kubernetes cluster or a database. The rectangles are components of our system in the execution environments, and the bounding box labeled “Deployment” shows the part of the system deployed to the cloud environment. The rectangles with folded corners are comments. The text within guillemot quotes («») are stereotypes indicating the type of component or deployment environment being represented.

This diagram illustrates how achieving resilience often requires using a collection of mechanisms together to address a resilience requirement. The TFX team would now continue with the other resilience requirements,

designing a set of mechanisms to achieve each element of resilience. One mechanism will often address a number of resilience scenarios (e.g., the client circuit breakers would also support resilience during the restart of a single system service scenario), but working through each scenario means that we know how the requirements of each are met and confirms the purpose of each mechanism within our system.

## Maintaining Resilience

A lot of the traditional focus in software architecture is at design time, architecting a suitable set of structures for the system, be that runtime functional, concurrency, data, or other structure. However, the architecture of a system is only valuable when implemented and operating in production, which is part of principle 5, *Architect for build, test, deploy, and operate*. It is encouraging to see the recent trend, encouraged by the DevOps movement, for software architects to take much more interest in the operability of their systems. In this section, we discuss some of the resilience concerns beyond the architecture of the software.

### Operational Visibility

In many of the systems we have encountered, there is very limited visibility of the system in production. It is quite common to find teams relying on their log files and perhaps some basic visualization of key runtime metrics, such as transaction throughput, CPU usage, and memory usage, along with a large amount of guesswork and intuition.

However, as the “recognize” element of our resilience model suggests, knowing what is going on in the production environment is an important part of achieving resilience. While the architecture of the system is just a small part of achieving resilience, getting a comprehensive view of any significant system is quite a complex undertaking and should be an architectural concern.

The key to operational visibility is monitoring, which is usually described as comprising metrics, traces, and logs. Metrics are the numeric measurements tracked over time, such as transactions per second, disk space used or value of transactions per hour. Traces are sequences of related

events that reveal how requests flow through the system. Logs are the timestamped records of events, usually organized by the system component creating them, the most common type of log being the simple textual message log we are all very familiar with. Monitoring is a kind of rapid feedback loop of the sort we introduced in [Chapter 2, “Architecture in Practice: Essential Activities,”](#) to provide us with the information needed to continually improve our system.

The problem with monitoring by itself is that it may well not be possible to do anything very useful with the data collected; that is, turning this into information may be rather difficult. How do these logs, metrics, and traces actually relate to what is going on (or went on) within the system? Actually, we need rather more than logs, metrics, and traces—we need to understand how the system works and the various states it can be in, and we must be able to correlate from the data we have collected to what state the system is in (or was in) and how it got there. This is a difficult problem, and this insight has recently resulted in a lot of interest in the observability<sup>[17](#)</sup> of the system, extending monitoring to provide insight into the internal state of the system to allow its failure modes to be better predicted and understood.

<sup>17</sup> Cindy Sridharan, *Distributed Systems Observability* (O’Reilly, 2018), provides an accessible introduction to this idea.

We don’t have a magical solution to this problem and don’t have space here to explore it in a lot of detail—that would be a book by itself. However, hopefully we have alerted you to the need to address this concern throughout the system life cycle. The good news is that many people are working on solutions, so there are both open source and commercial tools that can help solve parts, if not all, of the problem. This area moves so quickly that naming tools in a printed book is probably unhelpful and certainly unfair to those that we arbitrarily miss, but using one of the various directories (such as the Digital.ai [formerly Xebia] Periodic Table of DevOps Tools<sup>[18](#)</sup> or Atlassian’s DevOps Tools list<sup>[19](#)</sup>) will provide a useful starting point.

<sup>18</sup> <https://digital.ai/periodic-table-of-devops-tools>

<sup>19</sup> <https://www.atlassian.com/devops/devops-tools>

## Testing for Resilience

We never trust any piece of software until it has been tested, and this applies just as much to quality attributes such as performance, security, and resilience as it does to the functions of the system. As we discussed in [Chapter 2](#), integrating testing as an automated part of our delivery process, by shifting it left and using a **continuous testing** approach, is key to building confidence in our systems' quality attributes, and we should try to use this approach for resilience too.

Like many other quality attributes, resilience is quite difficult to test because there are so many different dimensions of it, and the resiliency of the system is quite context specific. What may cause a problem for one system may not be likely to happen to another, or even if it does, it might not cause a similar problem. Some creative thinking is usually needed to think through failure scenarios and create tests for them. It is also often difficult to create a safe environment for testing that is similar enough to the production environment to be truly representative in terms of scale and complexity. Even if we can create the runtime environment, replicating a sanitized version of the production dataset (or even copying the production dataset) may be a very difficult and time-consuming proposition. The other significant problem for many Internet-facing systems is how to generate a large enough synthetic workload to perform a realistic test.

All of these challenges mean that, while testing specific scenarios in the development cycle is still valuable, one of the most effective ways we know of to gain confidence in a system's resilience is to deliberately introduce failures and problems into it to check that it responds appropriately. This is the idea behind the famous Netflix Simian Army project,<sup>20</sup> which is a set of software agents that can be configured to run amok over your system and introduce all sorts of problems by killing VMs, introducing latency into network requests, using all the CPU on a VM, detaching storage from a VM, and so on. Approaches like this that inject deliberate faults into the system at runtime have become a key part of what is known as *chaos engineering*.<sup>21</sup> This approach can also be useful for testing other quality attributes, such as scalability, performance, and security.

<sup>20</sup> The original Simian Army project (<https://github.com/Netflix/SimianArmy>) has now been retired and replaced by similar tools from Netflix and others, such as the commercial Gremlin platform

(<https://www.gremlin.com>).

<sup>21</sup> Casey Rosenthal, Lorin Hochstein, Aaron Blohowiak, Nora Jones, and Ali Basiri, *Chaos Engineering* (O'Reilly Media, 2020). Also, *Principles of Chaos Engineering* (last updated May 2018). <https://www.principlesofchaos.org>.

Clearly, some care needs to be exercised, key stakeholders need to be aware of what is going on, and confidence needs to be built in nonproduction environments, but ultimately, nothing builds confidence in a system's resilience like seeing it deal with real failures even if they have been manufactured deliberately.

## The Role of DevOps

Historically, one of the reasons that the development and operations teams were in separate organizations was because they were seen as having different (and even opposing) goals. Development was goaled on making changes, operations was goaled on stability, which was usually interpreted as minimizing the number of times that things changed. Little wonder that the two groups often didn't get along!

One of the most powerful things about a DevOps approach is how it forms a unified, cross-functional team with the same goals: delivering value to operation reliably with the minimum cycle time.

DevOps is therefore very valuable when trying to build a resilient system because everyone now should see the reliability and stability of the production system as their goal, balancing this with the need to change it constantly. DevOps helps the entire team think about resilience much earlier and hopefully motivates people who are primarily building to focus on resilience features just as much as people who are mainly operating.

## Detection and Recovery, Prediction and Mitigation

We already discussed, under “[Operational Visibility](#),” how important it is to understand what is going on within a system and discussed the mechanisms that your system will need. Earlier in the chapter, we also discussed the importance of both backward-looking awareness and forward-looking prediction for achieving system resilience. Much of this information is well

outside the scope of architecture, but architecture has its role to play in ensuring that it is possible.

Achieving good operational visibility allows us to detect problematic conditions and recover from them. Analyzing our monitoring data from the past and considering what it is showing us and what may happen in the future allows us to predict possible future problems and mitigate them before they happen.

Exactly what you need depends on what your system does, its architecture, and the technologies it uses, and your options will be limited by the monitoring you can put in place and what you can realistically do with the information.

A good place to start is to ensure that you have enough monitoring information to detect whether each part of the system is operating normally via some sort of health check, which uses the monitoring information coming out of each component or (even better) implemented within the component that can be quickly queried with a simple network request. Once you can identify where you have a problem, you have the ability to recover from it as soon as it is apparent. As we noted in the section “[Architecting for Resilience](#),” ensuring that each component of your system has an explicit set of recovery mechanisms is a key building block for a resilient system.

Once you can detect and recover, the next level of sophistication is to be able to predict and mitigate. Prediction involves using your monitoring data to identify where a future problem might occur and doing so far enough ahead of its occurrence to allow mitigation. A classic approach is to track the historical trend of key metrics, such as throughput, peak loads, execution times for key operations, and so on, to spot trends heading in the wrong direction before they become critical. Tracing information and logs can also provide valuable input for prediction—for example, tracing may reveal transitory error or warning conditions that don’t appear to cause a problem today (perhaps because of robust retry logic) but, if a pattern emerges, might well point to a serious problem in the future.

## Dealing with Incidents

An unfortunate reality of life is that any operational system will suffer problems during its lifetime, and we commonly refer to operational problems in a production system as *incidents*.

Today, many organizations are moving away from traditional siloed support organizations and **Information Technology Infrastructure Library (ITIL)**-based<sup>22</sup> **service delivery management (SDM)** as they adopt a more DevOps way of working, where the skills need to develop *and* operate the system are merged into one cross-functional team.

<sup>22</sup> There are many books on ITIL, and an accessible explanation of ITIL in a DevOps context can be found in Matthew Skelton's "Joining Up Agile and ITIL for DevOps Success" (2014). <https://skeltonthatcher.com/2014/10/21/joining-up-agile-and-itil-for-devops-success>

We are extremely enthusiastic about the move to DevOps, having suffered the negative side effects of rigidly implemented traditional SDM ourselves for many years. However, an ITIL-based approach offers quite a few capabilities and activities that are still necessary in a DevOps world and need to be carried over or replaced. Incident management is one of them.

We've been involved in many production incidents, but we're not experts on incident management. We include this section, however, to remind you to play your part in setting up effective incident management for your system as part of achieving principle 5, *Architect for build, test, deploy, and operate*.

To achieve resilience, you need to be able to respond well to incidents, minimizing their impact and using each one as a learning opportunity. Effective incident response requires a well-structured, thoroughly rehearsed approach, with well-defined *roles* (e.g., incident command, problem resolution, and communication), a well-defined *process* for running the incident response, an effective pre-agreed set of *tools* for everyone to use (e.g., collaboration and knowledge management tools), and a well-thought-through approach to *communication* within the resolution team and outward from the resolution team.

More detailed advice on incident response can be found in many sources, but the chapters on emergency response and managing incidents in *Site Reliability Engineering: How Google Runs Production Systems* (referenced in "Further Reading") is a good place to start. In many organizations, it is

also worth having a basic knowledge of IT service management (ITSM) fundamentals, as found in frameworks such as ITIL, as using the same language as people who have been running incident management for many years can help to build some important organizational bridges and tap into deep pools of expertise in this area, such as running postmortems and root-cause analysis, which are skills that software development teams do not typically have much experience in.

## Disaster Recovery

Much of the literature and advice on designing and implementing resilient systems focuses on how to deal with localized failure, contain it, prevent things getting worse, and recover from it largely in place. This is valuable advice, and we've talked about a lot of these ideas and mechanisms in this chapter. However, before we leave the topic of resilience, it is important to consider larger-scale failures and the need for disaster recovery.

By *disaster*, we mean large-scale failure that may well be difficult to hide from users of our system and may involve significant effort to recover from. Examples in traditional enterprise computing are the loss of a data center or a major network problem. These events happen to cloud platforms too, with AWS, Azure, and Google Cloud Platform all having occasional significant outages of some of their availability zones, with huge resulting impact (e.g., the February 2019 AWS S3 outage, the GCP compute and storage problems in November 2019, and Azure's regional outage in September 2018 and capacity problems in March 2020).

The possibility of failures of this sort mean that we need to consider how they would affect us and must have a plan in place to mitigate such outages if we judge that the risk and likelihood of their occurring justifies the effort and costs involved.

Recovery from this sort of large-scale failure is termed **disaster recovery** and often involves manual effort as well as automated mechanisms. Options in the traditional enterprise world typically involve having spare capacity and standby system instances ready in data centers that are physically distant from each other. In the cloud world, it involves using the resiliency zones of the cloud provider to allow for complete failover from one area of

the world to another or even being able to restore service for our system on another cloud provider's platform.

As is immediately obvious, disaster recovery is complex and can cost significant amounts of time and money to provide reasonably comprehensive protection from unlikely but perfectly possible events. It is our job as software architects to balance these forces, identify an acceptable level of disaster recovery investment and implementation, and explain the tradeoffs to our key stakeholders to ensure that they agree with their implications.

In the example of TFX, the team must consider how TFX will survive a significant outage or problem in the cloud platform (e.g., by designing a mechanism to allow failover to a surviving availability zone) and must understand from their business and customers how significant an outage would be in exceptional situations. They can then work with their business to decide how much time and effort should be expended and what degree of outage is likely to be acceptable in an extreme case.

## Summary

The Internet era has raised expectations for system availability and reliability to the point that many systems are now expected to be always on. Our historical approach to achieving availability, using enterprise high-availability technology, such as clustering and data replication, can still be a good solution for some systems. However, as our systems grow in complexity, are decomposed into smaller and more numerous components, and as they move to public cloud environments, traditional high-availability approaches start to show their weaknesses.

In recent years, a new approach to achieving availability and reliability has emerged from the lessons of the early pioneers of Internet-scale systems. This approach still uses many of the fundamental techniques of data replication and clustering but harnesses them in a different way. Rather than externalizing availability from systems and pushing it into specialist hardware or software, architects today are designing systems to recognize and deal with faults throughout their architecture. Following decades of engineering practice in other disciplines, we term these systems *resilient*.

The architecture of a resilient system needs to include mechanisms to quickly *recognize* when problems occur, *isolate* problems to limit their impact, *protect* system components that have problems to allow recovery to take place, *mitigate* problems when system components cannot recover themselves, and *resolve* problems that cannot be quickly mitigated.

We have found that, in common with other quality attributes, scenarios are a good way to explore the requirements for resilience in a system and provide a structure for architects and developers to work through in order to ensure that their system will provide the resilience required.

A number of proven architectural tactics can be applied to create resilience mechanisms in the architecture of our systems, and we have discussed several of the more important ones in this chapter. However, it is also important to remember that achieving resilience requires focus on the entire operational environment—people and processes as well as technology—in order to be effective.

Finally, a culture of measuring and learning allows a software product team to learn from their successes and failures, continually improve their level of resilience, and share their hard-won knowledge with others.

## Further Reading

Resilience is a broad field, and we have introduced only some of its important concepts in this chapter. To learn more, we recommend starting with some of the references we list here.

- Michael Nygard's book *Release It! Design and Deploy Production-Ready Software*, 2nd ed. (Pragmatic Bookshelf, 2018) provides a rich seam of experience-based advice on designing systems that will meet the demands of production operation, including improving availability and resilience. A number of our architectural tactics come directly from Michael's insights.
- A good text on classical resilience engineering is *Resilience Engineering in Practice: A Guidebook* (CRC Press, 2013) by Jean Pariès, John Wreathall, and Erik Hollnagle, three well-known figures in that field. It does not talk about software specifically but provides a lot of useful advice on organizational and complex system resilience.

- A good introduction to observability of modern systems is Cindy Sridharan’s *Distributed Systems Observability* (O’Reilly, 2018). A good introduction to chaos engineering can be found in *Chaos Engineering: System Resiliency in Practice* (O’Reilly, 2020) by former Netflix engineers Casey Rosenthal and Nora Jones.
- Two books from teams at Google that we have found valuable are *Site Reliability Engineering: How Google Runs Production Systems* (O’Reilly Media, 2016) by Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy, which is particularly useful for its discussion of recognizing, handling, and resolving production incidents and of creating a culture of learning using postmortems; and *Building Secure and Reliable Systems* by Ana Oprea, Betsy Beyer, Paul Blankinship, Heather Adkins, Piotr Lewandowski, and Adam Stubblefield (O’Reilly Media, 2020).
- You can find useful guidance on running learning ceremonies known as *retrospectives* in Ester Derby and Diana Lawson’s *Agile Retrospectives: Making Good Teams Great* (Pragmatic Bookshelf, 2006). Aino Vonge Corry’s *Retrospective Antipatterns* (Addison-Wesley, 2020) presents a unique summary of what *not* to do in a world gone agile.
- The following offer abundant information on data concerns.

Pat Helland, “Life beyond Distributed Transactions: An Apostate’s Opinion,” in *Proceedings of the 3rd Biennial Conference on Innovative DataSystems Research (CIDR)*, 2007, pp. 132–141.<sup>23</sup>

Ian Gorton and John Klein, “Distribution, Data, Deployment: Software Architecture Convergence in Big Data Systems,” *IEEE Software* 32, no. 3 (2014): 78–85.

Martin Kleppman, *Designing Data Intensive Systems* (O’Reilly Media, 2017).

- Finally, an idea that seems to come and go in the software resilience area is *antifragility*. This term appears to have been coined by the well-known author and researcher Nassim Nicholas Taleb in *Antifragile*:

*Things That Gain from Disorder* (Random House, 2012). The core idea of antifragility is that some types of systems become stronger when they encounter problems, examples being an immune system and a forest ecosystem that becomes stronger when it suffers fires. Hence, antifragile systems are by their nature highly resilient. The original book doesn't mention software systems, but several people have applied this idea to software, even proposing an antifragile software manifesto; see Daniel Russo and Paolo Ciancarini, "A Proposal for an Antifragile Software Manifesto," *Procedia Computer Science* 83 (2016): 982–987. In our view, this is an interesting idea, but our observation is that a lot of the discussion we've seen of it in the software field seems vague, and it is hard to separate these proposals from better established ideas (e.g., chaos engineering and continuous improvement).

<sup>23</sup> Also available from various places, including <https://www-db.cs.wisc.edu/cidr/cidr2007/papers/cidr07p15.pdf>

# Chapter 8. Software Architecture and Emerging Technologies

*As an architect you design for the present, with an awareness of the past for a future which is essentially unknown.*

—Norman Foster

Many organizations have teams that focus on testing and applying emerging technologies in order to gain experience with those technologies before using them on other projects. Other organizations attempt to use emerging technologies without prior experience on their new development projects, sometimes with adverse consequences on delivery schedules and delivery risks. If you asked the teams involved with emerging technologies to list their key concerns, an architecture-led approach may not appear at the top of the list. The notion that software architecture may help them deal with the challenges associated with applying new technologies may not be something that they would agree with.

Yet IT professionals with architecture skills are good at identifying opportunities offered by new technologies that might not be obvious. They tend to have deep technical knowledge and can help with the art of the possible. For example, projects leveraging popular new technologies such as artificial intelligence (AI) and machine learning (ML) tend to be data intensive. A number of lessons from advanced analytics and big data architectures can be applied to those projects to improve velocity and reduce implementation risk.

Some emerging technologies, such as shared ledgers, have profound architecture implications, yet architecture most likely is not a consideration when those technologies are evaluated. Shared ledger implementations have significant architectural challenges such as latency, throughput, and access to external information. Those challenges are sometimes ignored because no one is thinking about an overall architecture.

Mature organizations often partner with startups to boost innovation, and as a result new technologies are quickly introduced into those organizations. How they fit into the organization's existing architecture is usually not a concern, and our experience shows that architects' attempts to integrate those technologies into their architecture are often unsuccessful.

This chapter highlights the value of an architecture-driven approach when dealing with emerging technologies. We present a brief overview of three key emerging technologies: AI, ML, and deep learning (DL). We discuss how an architecture-led approach can be used to implement those technologies within the context of the TFX case study. We then turn our attention to another key emerging technology, shared ledgers, and explain how architecture skills and thinking would support its implementation within the TFX context.

Recall that in our case study, the team is working for a software company that is building an online service, called Trade Finance eXchange (TFX), that will provide a digital platform for issuing and processing letters of credit (L/Cs). We discuss these emerging technologies in practical rather than theoretical terms, in the context of the TFX case study. We particularly focus on an architectural point of view and provide a few useful architecture tools to deal with the next emerging technology trend. For example, see the sidebar “[When Nontechnical Stakeholders Want to Introduce New Technologies](#)” in this chapter.

## Using Architecture to Deal with Technical Risk Introduced by New Technologies

One of the benefits of adopting an architecture-led approach when a team attempts to use emerging technologies is that the architecture can be used to plan and communicate with stakeholders. Sound planning and clear communication significantly reduce implementation and technology risk, which is a key consideration on these type of projects.

We have found that projects that deal with new technologies are more risky than other new development or other architecture projects. Therefore, activities that minimize risk, such as focus on architectural decisions and tradeoffs and careful evaluation of new technologies, are beneficial.

Architecture narrows the evaluation context so evaluation scope can be tailored to the intended use and provide the best return on the evaluation investment.

Finally, when dealing with emerging technologies, it is important to do hands-on prototyping to gain knowledge about that technology before proceeding to architecture and design activities.

## Brief Introduction to Artificial Intelligence, Machine Learning, and Deep Learning

The purpose of this section is to introduce the terms and definitions needed to discuss how these technologies can be applied in the TFX case study. Agreement on precise definitions of AI, ML, and DL can be hard to come by. Fortunately, we need only brief descriptions for this chapter. Readers interested in learning more about AI, ML, and DL should refer to specialized books such as the ones mentioned in the “[Further Reading](#)” section.

AI is generally described as intelligence demonstrated by machines, by imitating thinking functions usually associated with the human brain, such as learning and problem solving.<sup>1</sup> ML can be characterized as augmenting software systems with statistical models in order to perform a specific task without using explicit instructions.<sup>2</sup> DL is generally considered to be a subset of ML and uses artificial neural networks.<sup>3</sup>

<sup>1</sup> Wikipedia, “Artificial Intelligence.” [https://en.wikipedia.org/wiki/Artificial\\_intelligence](https://en.wikipedia.org/wiki/Artificial_intelligence)

<sup>2</sup> Wikipedia, “Machine Learning.” [https://en.wikipedia.org/wiki/Machine\\_learning](https://en.wikipedia.org/wiki/Machine_learning)

<sup>3</sup> Wikipedia, “Deep Learning.” [https://en.wikipedia.org/wiki/Deep\\_learning](https://en.wikipedia.org/wiki/Deep_learning)

We also need to introduce the ML model concept, which is important from an architectural perspective. This concept may be used in a confusing way in some projects, as it is generally used in reference to both code and ML parameters, such as weights. We use the Keras<sup>4</sup> definition in this chapter, which defines *model* as including both, but they are distinct (see Keras documentation about saving models).<sup>5</sup> In architecture terms, the model is a

new type of artifact that the software architecture, as well as the DevSecOps processes and pipeline, must accommodate.

<sup>4</sup> Keras ML Framework. <https://keras.io>

<sup>5</sup> Keras FAQ, “What Are My Options for Saving Models?”  
[https://keras.io/getting\\_started/faq/#what-are-my-options-for-saving-models](https://keras.io/getting_started/faq/#what-are-my-options-for-saving-models)

## Types of Machine Learning

Given a set of inputs and their corresponding outputs, an ML process “learns” to map inputs to outputs. ML processes are used when outputs cannot be derived from inputs using a set of rules. The most common types of ML algorithms are supervised learning, unsupervised learning, and reinforcement learning.

### *Supervised Learning*

Supervised learning uses labeled training datasets to learn model parameters to perform classification of operational datasets.

*Classification* is an example of supervised learning. The goal of a classification process is to divide its inputs into two or more classes. In the context of the TFX case study, it is estimated that, in 2016, only 64 percent of L/C document presentations (e.g., presentations of bills of lading, and insurance documents) worldwide, complied upon first presentation; in other words, they did not have any discrepancies.<sup>6</sup> This situation creates a serious problem for the issuing bank when discrepancies are overlooked and payment is made. Classification could be used to predict whether a document with discrepancies is likely to be accepted or rejected. This is an example of a classification problem because the classes we want to predict are discrete: “document likely to be accepted” and “document likely to be rejected.”

<sup>6</sup> John Dunlop, “The Dirty Dozen: Trade Finance Mythology,” *Trade Finance Global* (August 30, 2019). <https://www.tradefinanceglobal.com/posts/international-trade-finance-mythology-the-dirty-dozen>

### *Unsupervised Learning*

This type of ML process is applied to find structure in datasets consisting of input data without labeled responses. Unsupervised learning derives model parameters directly from the operational data without using labeled training datasets. Cluster analysis is an example of unsupervised learning and finds hidden patterns or groupings in data. For example, in the context of our case study, imagine having access to data about all products being imported into a given country as well as data about the importers and the exporters. Using that data, cluster analysis could find patterns and identify that, for example, smaller companies are more likely to import products from established exporters in Asia, whereas larger companies may like to deal with smaller suppliers (this is, of course, a totally fictitious example). This information can help the TFX system predict which company is more likely to import products from which exporter. This in turn could help banks customize their L/C offerings and market those offerings to their customers. We will meet unsupervised learning again when we review the training of models as part of the TFX case study.

## ***Reinforcement Learning***

In reinforcement learning, a computer program interacts with a dynamic environment in which it must perform a certain goal (e.g., driving a vehicle or playing a game against an opponent). The program is provided feedback in terms of rewards and punishments as it navigates its problem space. However, we have not found a good use of reinforcement learning for the TFX initial requirements and therefore further discussion of this type of ML is beyond the scope of this book.

## **What About Deep Learning?**

DL leverages artificial neural networks, commonly referred to simply as *neural networks* by AI and ML practitioners. Using neural networks with ML methods, and especially with supervised learning, DL extracts patterns and detects trends that may be too complex to be identified using other approaches. How neural networks work isn't relevant to most people using them, as they are implemented using prebuilt libraries and services.

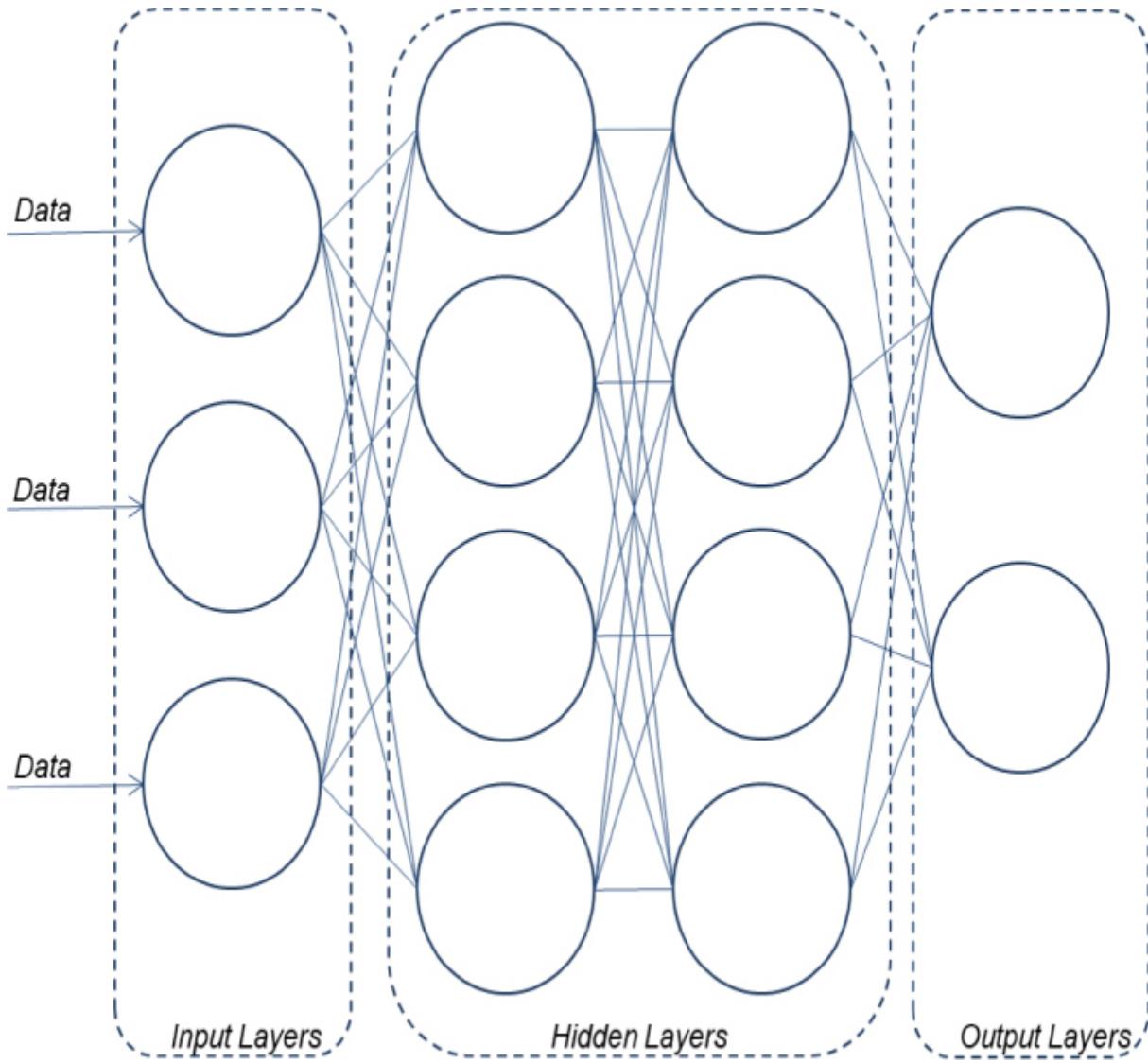
However, we are presenting a very brief overview of this technology in the following paragraphs and encourage readers who want to know more about this topic to consult Russel and Norvig.<sup>7</sup>

<sup>7</sup> Please refer to Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. (Pearson, 2020) for an in-depth discussion of neural networks and their operation.

Neural networks consist of a large, highly interconnected graph of very simple statistical steps, known as *neurons*. Neural networks often include the following three kinds of layers:

- *Input layers*: The function of the neurons in the input layers is to acquire data for processing by the neural network.
- *Hidden layers*: The neurons located in the intermediate layers are responsible for the main processing of the neural network.
- *Output layers*: The neuron in the output layers produce the neural network results.

[Figure 8.1](#) provides an overview of a typical neural network.



**Figure 8.1** Neural network overview

In a neural network, data travels from the input layers to the output layers, going through a layer several times if required. An additional capability of a neural network is the ability to learn from training data. A neural network evolves and adapts on the basis of training data that it processes during the training process and compares its output to the desired results. This is achieved by adjusting the connections between neurons and layers until the accuracy of the network reaches a desired level. The neural network then uses that knowledge to make predictions using actual data as its input.

# Using Machine Learning for TFX

This section examines ML from an architectural perspective. We look at the types of problems that ML is well suited to solve, the prerequisites for applying ML, and new architecture concerns that ML introduces.

## Types of Problems Solved by ML, Prerequisites and Architecture Concerns

As we saw earlier in this chapter, ML can be depicted as the application of statistical models to enable software systems to perform a specific task without explicit instructions. Overall, ML is good at solving pattern-matching problems with “noisy” data. The techniques vary on how to train the system to deal with recognizing patterns in this data, but the basic type of problem that it helps to solve is an advanced conditional decision process that involves (1) recognizing a pattern and (2) choosing a related action. Narrowing this statement to financial services, some of the areas where ML could be used for TFX include

- *Document classification:* Ability to digitize the document contents using optical character recognition (OCR) and then use natural language processing (NLP) or other text processing to classify the document contents.
- *Chatbots and conversational interfaces for customer service:* Ability to automate (or partially automate) some of the interactions between L/C specialists at the bank and their customers.
- *Data entry:* Ability to automate some of the data entry tasks associated with L/C processing.
- *Advanced analytics:* ML can be used to provide advanced analytics, such as predicting whether an importer is likely to do more L/C business with the bank.

We focus on document classification and chatbots in this chapter. Following an architecture-led approach, we need to understand the ML prerequisites in order to effectively use it in those areas. Keeping in mind that ML applies statistical models to large amounts of data, we need to be able to architect that data, which implies that data-intensive architecture principles and

tactics, such as the ones used for big data, need to be leveraged to create an effective solution.

Lessons from advanced analytics and big data architectures can be applied to AI and ML projects to improve project velocity and reduce risk. Some architectural decisions are likely to be already made for the team. For example, a decision has been made to use a NoSQL database as part of the TFX architecture. Each NoSQL product is associated with several quality attribute tradeoffs, such as performance, scalability, durability, and consistency. One of the best practices when elaborating an architecture that meets quality attribute requirements is to use a series of appropriate architectural tactics. As stated earlier in this book, a tactic is a design decision that impacts the control of one or more quality attribute responses. Tactics are often documented in catalogs to promote reuse of this knowledge among architects.<sup>8</sup>

<sup>8</sup> See [https://quabase.sei.cmu.edu/mediawiki/index.php/Main\\_Page](https://quabase.sei.cmu.edu/mediawiki/index.php/Main_Page) for a catalog of software architecture tactics for big data systems.

In addition, supervised ML models need to be trained using training and validation datasets, which are distinct from the operational data that will be used by the ML models to make predictions once they have been trained. Availability of training and validation datasets is a prerequisite to using supervised ML, for example, for use cases such as document classification and chatbots.

ML also introduces new architecture concerns in addition to concerns associated with data-intensive systems. As is true for any data-intensive system, scalability, performance, and resiliency tactics apply to ML systems. In addition, data versioning of training and validation datasets is an important aspect for architecting ML systems.

New concerns introduced by ML systems include the ability to track the provenance of the data, metadata management, and the observability of ML task performance. Data provenance documents the systems and processes that affect the data and provides a record of its creation. Metadata<sup>9</sup> management deals with the handling of data that describes other data; see “[Data Ingestion](#)” later in the chapter for more details on how those concepts are used in practice.

<sup>9</sup> See Chapter 3, “Data Architecture,” for more information on metadata.

Tracking the provenance of the data and managing metadata are important for the overall ML process, especially for preparing data for processing by the ML model, so special attention needs to be paid to the data architecture. In addition, performance and scalability architecture considerations are very important. Being able to observe the performance of ML tasks is key to monitoring the performance of the overall ML system. Finally, it is important to ensure that models created by data scientists, especially in-house models, are validated as follows before being used in a production environment:

- They should be thoroughly inspected for any security issues.
- They should have been appropriately tested and debugged.
- Their results should be explainable, meaning that they can be understood by humans. Their bias should be measured and understood.

The next section explains how document classification could be used for TFX. Later, we discuss how chatbots could be integrated into the TFX platform.

## Using Document Classification for TFX

One of the TFX business goals is to automate the document verification process as much as possible at payment time to eliminate several manual, lengthy, and potentially error-prone tasks. As mentioned earlier, it is estimated that up to 70 percent of L/C documents contain discrepancies upon first presentation to the bank. Those documents are mainly paper based, and discrepancies are not always noticed by the L/C specialist who manually verifies them against the requirements of the L/C. Discrepancies may not even be noticed until the importers receive the goods and discover that they don't exactly match what they ordered. This may trigger a long and painful litigation process.

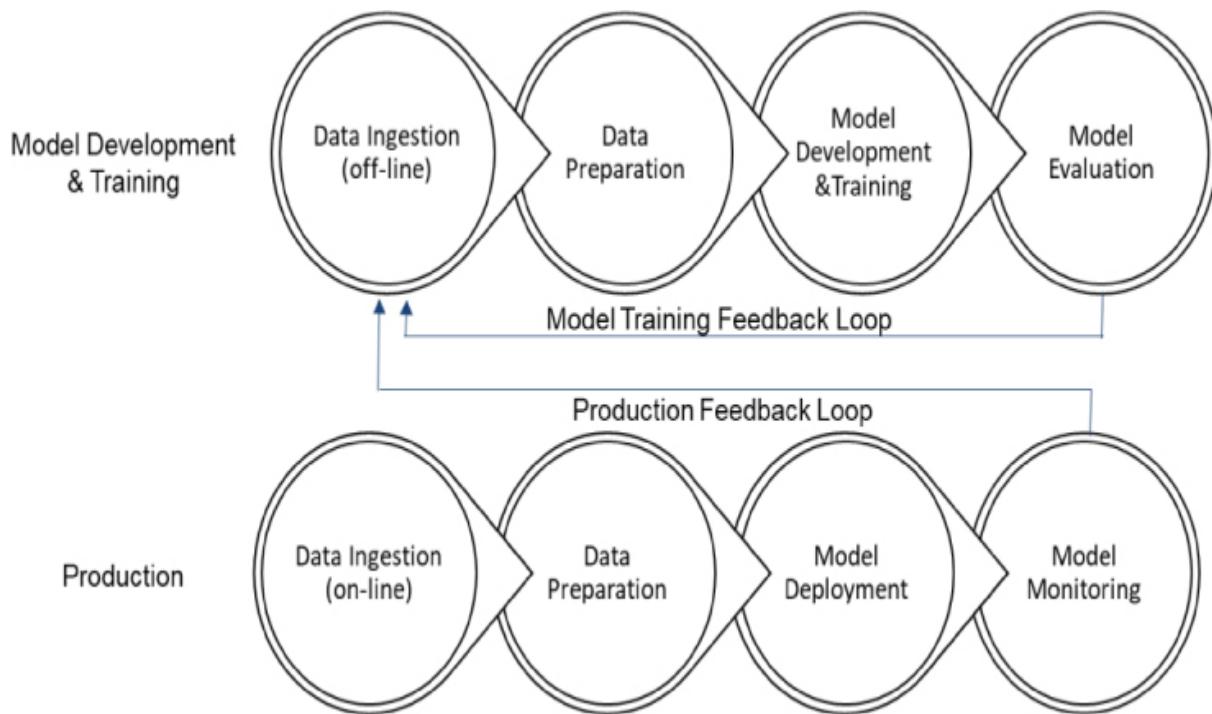
Digitizing the contents of the documents using OCR technology and matching their digitized contents to the requirements specified in the L/C would be part of the solution. However, simple text matching would be insufficient to solve this problem. Specifically, the description of the goods from the documents often does not exactly match what's outlined in the

L/C. Abbreviations are frequently used—for example, “LTD” instead of “Limited” or “Co” instead of “Company.” Spelling mistakes are common given translations between export country and import country languages—for example, “mashine” instead of “machine” or “modle” instead of “model.” Synonyms may even be used—for example, “pants” instead of “trousers.” If the document uses an abbreviation, has spelling mistakes, or uses synonyms, no discrepancy is usually called. Given those challenges, trying to match the results of the OCR process to the L/C requirements using a traditional IT process would not yield good results.

To address this issue, we need to leverage ML so that TFX is able to “read” and “understand” the text generated by the OCR utility from the documents. TFX could then turn their contents into meaningful information that could be used for the verification process, in essence mimicking what an L/C specialist does.

## ***TFX Architecture Approach***

Based on the ML prerequisites and architecture concerns, the TFX team designs and implements an architecture. They then separately design two processes (also known as *ML pipelines*) to include a document classification model in the TFX platform. [Figure 8.2](#) shows a high-level depiction of the ML pipelines. Note that those pipelines are not specific to TFX. Most ML projects would use similar ones.



**Figure 8.2** *ML pipelines: Model Development & Training and Production*

This architecture follows principle 5, “Architect for build, test, deploy, and operate.” When developing, training, and evaluating ML models, data scientists usually work in a sandbox development environment, which is distinct and separate from the traditional IT environments. Models, which are comprised of code and model parameters, need to be promoted from sandbox environments into IT testing and production environments. As mentioned earlier, ML models are a new type of artifact that the software architecture, as well as the DevSecOps processes and pipeline, must accommodate. In addition, the architecture uses principle 2, “Focus on quality attributes, not on functional requirements.” Specifically, it focuses on scalability, performance, and resiliency and applies the tactics discussed in this book. Overall, this architecture is a data-intensive architecture that deals with very large amounts<sup>10</sup> of data. As such, big data architecture principles and tactics, such as partitioning the database to distribute read load, need to be leveraged to create an effective architecture solution.

<sup>10</sup> Typically, multiple terabytes of data.

From an architectural perspective, the most significant components in the ML pipelines are the data ingestion, data preparation, model deployment,

and model monitoring.

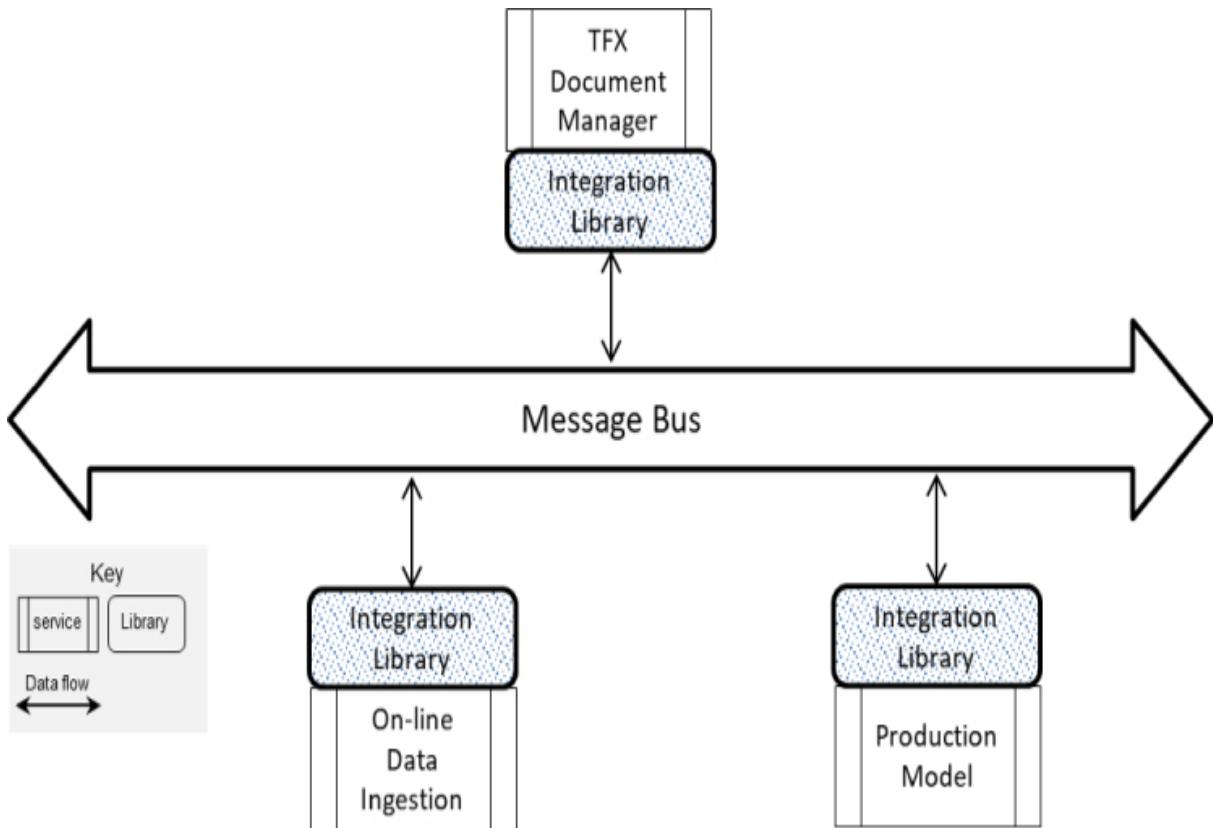
## ***Data Ingestion***

There are two types of data ingestion capabilities that need to be provided by the architecture: online ingestion and offline ingestion. Within the context of TFX, offline ingestion is used to collect documents in batch mode for training the models, as shown in [Figure 8.2](#). Online ingestion is used to collect documents used as an input for the models to run in production, an activity referred to as *model scoring*.

Two important concerns that need to be addressed by data ingestion are the ability to track the provenance of the data and metadata management. In order to track the provenance of the data, the first step in the ingestion process should be to store a copy of the data before making any changes to it. Because large amounts of data are involved in the offline ingestion process, using a NoSQL document database would be a good technical solution to implement this requirement, especially if the data is distributed over several database nodes. Another option would be to use object storage such as Amazon Simple Storage Service (Amazon S3),<sup>11</sup> but a NoSQL is a better option for TFX because it allows metadata to be attached to the data by storing each dataset as a document. Metadata describing the fields in the input datasets needs to be captured at this stage.

<sup>11</sup> Also see “What Is Amazon Machine Learning?” <https://docs.aws.amazon.com/machine-learning/latest/dg/what-is-amazon-machine-learning.html>

Because performance and scalability are important quality attributes, communication between TFX components and the online data ingestion component should be asynchronous in order not to create a bottleneck when production documents are being processed by the models. This could be implemented using a message bus (see [Figure 8.3](#)). Note that the documents are scanned by the TFX Document Manager before ingestion by the Online Data Ingestion component.



**Figure 8.3** Document classification components integration with TFX

### **Data Preparation**

The next significant component from an architectural perspective is data preparation, which is different for the development and training pipeline than for the production pipeline (refer to [Figure 8.2](#)). Data preparation in development and training includes data cleansing and feature engineering of data collected by the offline ingestion process. Data can be automatically inspected and cleansed by a data preparation component. However, some manual intervention may be required as well. For example, the team may need to manually validate that the document images are clear enough to be processed by the OCR utility with good results. Some documents may need to be imaged again to improve the results of the OCR process.

Feature engineering is a design-time activity that creates and/or selects how the dataset is input to the ML model. In feature engineering, another data preparation component extracts features<sup>[12](#)</sup> from raw data in order to improve the performance of ML algorithms.<sup>[13](#)</sup> Once those steps are

completed, data collected by the offline collection process is manually labeled with the expected prediction results.

<sup>12</sup> In ML, features are discrete variables that act as an input for the system.

<sup>13</sup> Wikipedia, “Feature Engineering.” [https://en.wikipedia.org/wiki/Feature\\_engineering](https://en.wikipedia.org/wiki/Feature_engineering)

Data preparation in production is limited to automatic data inspection and cleansing, without any manual intervention. It also includes a process of *feature generation*, or *feature extraction*, that occurs in the production pipeline. This process is complementary to the feature engineering process and is related to embedding, but that concept starts to get deeper into ML technology and is beyond the scope of this book.

In the development and training pipeline, input documents may be separated into two datasets: a training dataset, used to train the ML models, and a validation dataset, used to validate that the models’ predictions are acceptable. An alternative approach is cross-validation that uses one dataset for both purposes. Either way, preparing a training set is a time-consuming task, as the size of the training set drives the accuracy of the neural network: using too few documents results in an inaccurate network. Unfortunately, there is no hard and fast rule to determine the optimal size of the training set, and the data preparation portion of the ML pipeline may take a significant amount of time until acceptable model accuracy is reached while training the model.

## ***Model Deployment***

A good way to implement document classification for TFX is to use an NLP neural network. Creating a neural network for TFX using languages such as R or Python may be tempting, but leveraging open source software from the experts, such as software from the Stanford NLP Group,<sup>14</sup> is a better way to implement the model. Given the architecture tradeoffs involved in the first option, such as maintainability, scalability, security, and ease of deployment, leveraging prepackaged reusable software is a better option (see [Table 8.1](#)). (The complete TFX Architecture Decision Log is in [Appendix A](#).)

<sup>14</sup> See <https://stanfordnlp.github.io/CoreNLP>

**Table 8.1** Decision Log Entry for TFX Document Classification Model

Name	Type	ID	Description	Options	Rationale
TFX document classification model	ML	AD-ML-1	TFX will leverage prepackaged reusable software for document classification	Option 1, create a neural network for TFX using languages such as R or Python Option 2, leverage open source software from the experts, such as software from the Stanford NLP group	Given the architecture tradeoffs involved in the first option, such as maintainability, scalability, security, and ease of deployment, leveraging prepackaged reusable software is a better option

As mentioned earlier, this model is used in a sandbox environment for training and evaluation purposes. However, once the model is adequately trained and provides acceptable predictions, it needs to be promoted to a set of IT environments for eventual use in production. The ML model artifact, including the model code and parameters, needs to be integrated into the TFX DevSecOps pipeline. This integration includes packaging the ML model artifact files and validating the model using an automated testing suite before deployment. It also includes testing the model for potential security issues. Packaging up the model and then deploying it into an environment is referred to as *model serving* (refer to [Figure 8.2](#)).

The model code is deployed as one of the TFX services in the production environment (refer to [Figure 8.3](#)). The model parameters are cached for scalability and performance, using caching as discussed in [Chapter 5](#), “Scalability as an Architectural Concern.”

## ***Model Monitoring***

Once the model is implemented, its performance needs to be closely monitored. Performance has two meanings in an ML context: (1) model classification, precision, and recall or (2) throughput and latency. From an architectural perspective, it is important to use scenarios to define quality

attribute requirements such as performance and not try to satisfy a vague requirement such as “the model will have acceptable performance.”

Monitoring the performance of the model for document classification accuracy is done by logging key information for each run of the model, including model identification, dataset used for the model, and predicted versus actual results. This information is collected and analyzed on a regular basis. The model may need to be retrained with new data and redeployed if its accuracy degrades over time. This would happen if the characteristics of the input data no longer match those of the training set—for example, if a new type of goods with different abbreviations has been added to the input data.

Monitoring the model for throughput and latency is achieved by using the same tactics as the ones we discussed in [Chapter 5](#). Specifically, it involves logging information about when a document was submitted into the TFX ML production pipeline, how long it took to prepare it for processing by the ML model, and the duration of the model run for that document. That information is collected and analyzed on a regular basis, and the production environment for the model is tuned as necessary. In addition, alerts are generated when the overall processing time for a document from submission to classification exceeds a given threshold.

## ***Common Services***

The TFX ML architecture leverages several common components from the TFX architecture to support the ML pipeline processes. Those components include the following:

- Data component, including metadata and data provenance
- Caching component
- Logging and log processing component
- Instrumentation component
- Alerts and notifications component

## ***Benefits of an Architecture-Led Approach***

As mentioned earlier in this chapter, one of the benefits of adopting an architecture-led approach to implementing ML is that the architecture can be used to plan development and communicate with stakeholders.

Leveraging an architecture for planning and communication significantly reduces implementation and technology risk. Activities such as focusing on architectural decisions and tradeoffs, as well as careful evaluation of new technologies, are very beneficial on ML projects.

An architecture-led approach could foster the adoption of ML in the enterprise. As we saw earlier, quality attributes such as security, scalability, and performance, in terms of classification accuracy as well as throughput and latency, are critical considerations when architecting a software system that uses an ML model. Successfully running a model in a sandbox environment does not guarantee that the same model will perform well in a production environment. In addition, **continuous integration** and deployment processes need to include the ML models and conventional code and configuration. In summary, ML models and components should be treated like any other software artifact. They do not need to be special just because they are a new technology.

We shouldn't isolate new technologies but integrate them into our existing systems. We should consider including ML components in our applications. For example, we could include AI assistants in our traditional IT architectures where they make sense. For TFX, an AI assistant, also known as a *chatbot*, could help an L/C specialist handle customer queries, which leads us to the second area of ML applicability for TFX covered in this chapter. The next section discusses how chatbot technology could be implemented within the TFX architecture.

## Implementing a Chatbot for TFX

What exactly is a chatbot? It is simply a software service that can act as a substitute for a live human agent, by providing an online chat conversation via text or text-to-speech.<sup>15</sup>

<sup>15</sup> See Wikipedia, “Chatbot.” <https://en.wikipedia.org/wiki/Chatbot>

This service is a good candidate for inclusion in the scope of TFX. L/C departments often receive a high volume of telephone calls and emails from

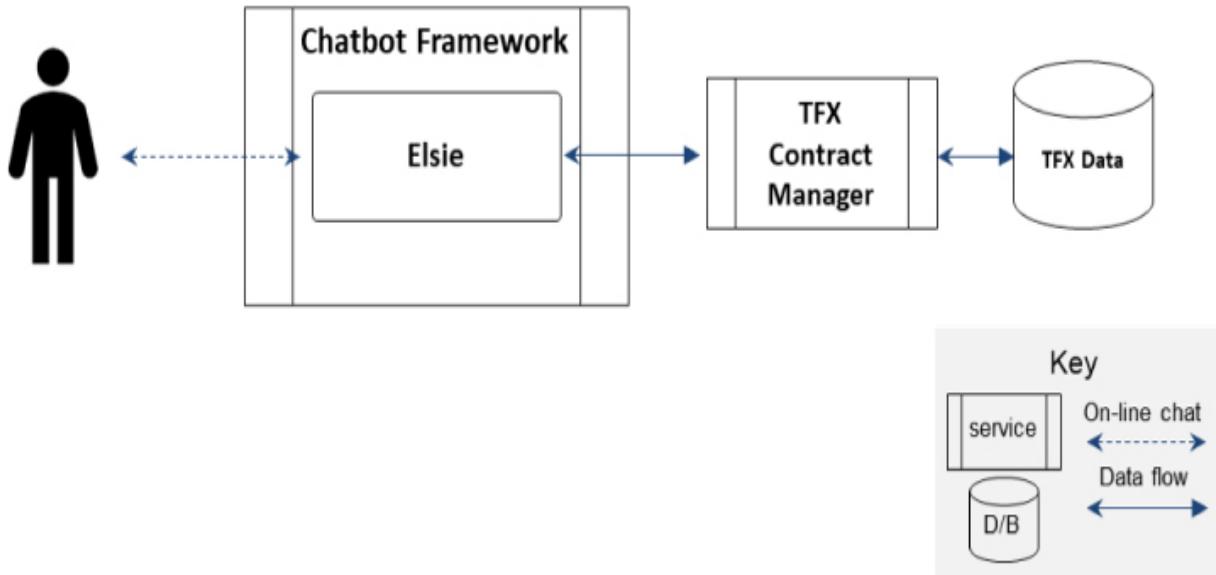
importers asking, for example, when their L/C will be issued and from exporters who want information such as when they will be paid. As a result, L/C specialists spend valuable time researching and answering questions. Automatically handling a portion of those queries by using a chatbot would make the L/C departments more productive. Let us walk through a hypothetical scenario of implementing a chatbot for TFX following Continuous Architecture principles.

### ***Elsie: A Simple Menu-Driven Chatbot***

Because the TFX team does not have any experience with chatbots, they decide to start by implementing a simple TFX chatbot. They apply principle 4, “Architect for change—leverage ‘the power of small,’” and create a modular chatbot design that can be extended as the need arises. They keep in mind principle 3, “Delay design decisions until they are absolutely necessary,” and do not try to incorporate too many requirements—neither functional nor quality attribute requirements—to early in the process.

Following this approach, they decide to use an open source, reusable chatbot framework.<sup>16</sup> This framework can be used to implement a range of customer service chatbots, from simple menu-based chatbots to more advanced ones that use natural language understanding (NLU). Because they want to leverage the power of small, they first create a menu-based single-purpose chatbot capable of handling straightforward queries from importers (see [Figure 8.4](#)).

<sup>16</sup> Rasa Open Source is an example of such a framework. <https://rasa.com>



**Figure 8.4** Elsie’s initial architecture: single purpose and menu based

Elsie,<sup>17</sup> the aptly named TFX, or L/C, chatbot, interfaces with the TFX Contract Manager using the TFX APIs. Elsie presents a simple list of options to the importers as follows:

<sup>17</sup> Elsie rhymes with L/C, our abbreviation for letters of credit.

1. Query L/C status by importer name
2. Query L/C status by exporter name
3. Query L/C status by L/C number
4. Frequently asked questions (FAQs)
5. Help
6. Chat with an agent
7. Quit

Using a smartphone, a tablet, a laptop, or a desktop computer, an importer can select an option from Elsie’s simple menu of options and obtain information. The team can rapidly implement this simple design both as a mobile app and as a browser-based app running in a popup window, using the open source chatbot framework.

The architecture of this chatbot follows Continuous Architecture principles and specifically uses principle 2, “Focus on quality attributes, not on functional requirements.” Security considerations (refer to [Chapter 4](#), “Security as an Architectural Concern”) are taken into account in the design. For menu options 1, 2, and 3, Elsie ensures that an importer is authorized to access the L/C information that Elsie is retrieving and that the importer cannot access L/C data related to another importer. This means that the importers need to enter their credentials before being able to access the TFX chatbot. Elsie retrieves the importer’s credentials and keeps them in a session state storage area in order to authenticate the importer and validate access to the TFX information. This does not apply to accessing FAQs (menu option 4).

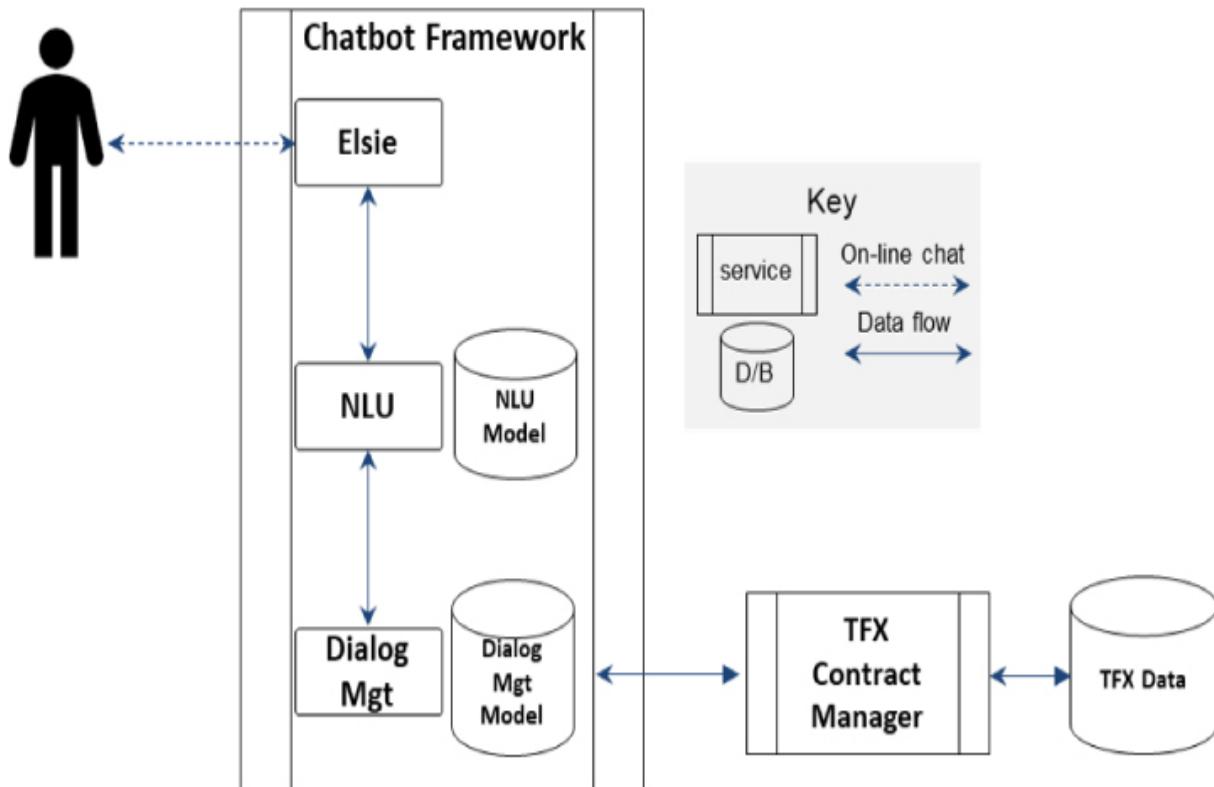
Initial user tests go well. There are no concerns at this time with performance or scalability. The business testers are pleased with Elsie’s capabilities. However, they are concerned with the limitations of a simple menu-based user interface. As more capabilities are added to the TFX chatbot, such as the ability to provide information beyond a simple L/C status, the menu-based interface becomes harder to use. It displays too many options and becomes too long to be shown entirely on a smartphone screen or in a small popup window. In addition, business testers would like to jump from one menu option to another without having to return to the main menu and navigate through submenus. They would like to converse with Elsie in a more familiar way, using natural language.

## *A Natural Language Interface for Elsie*

The open source chatbot framework used by our team for Elsie’s first implementation includes support for NLU, so it makes sense to continue using this framework to add NLU to Elsie’s capabilities. Using NLU transforms Elsie into an ML application, and some of the ML architecture considerations we discussed in the previous section on document classification indirectly apply here as well.

Specifically, the team needs to implement a simplified ML pipeline and update the architecture of the TFX chatbot (see [Figure 8.5](#)). As we saw earlier, data ingestion and data preparation in offline mode for training data are two architecturally important steps, as are model deployment and model performance monitoring. Model monitoring for language recognition

accuracy as well as throughput and latency are especially important because TFX platform users do not want to converse with a chatbot that either does not understand their questions or is slow in responding. Business users use certain trade finance terms, which Elsie gets better at understanding over time.



**Figure 8.5** Elsie’s NLU architecture

The new architecture leverages ML and includes two models that need to be trained in a sandbox environment and deployed to a set of IT environments for eventual use in production (refer to [Figure 8.5](#)). Those models include an NLU model, used by Elsie to understand what the importer wants her to do, and a dialog management model, used to build the dialogues so that Elsie can satisfactorily respond to the messages.

### ***Improving Elsie’s Performance and Scalability***

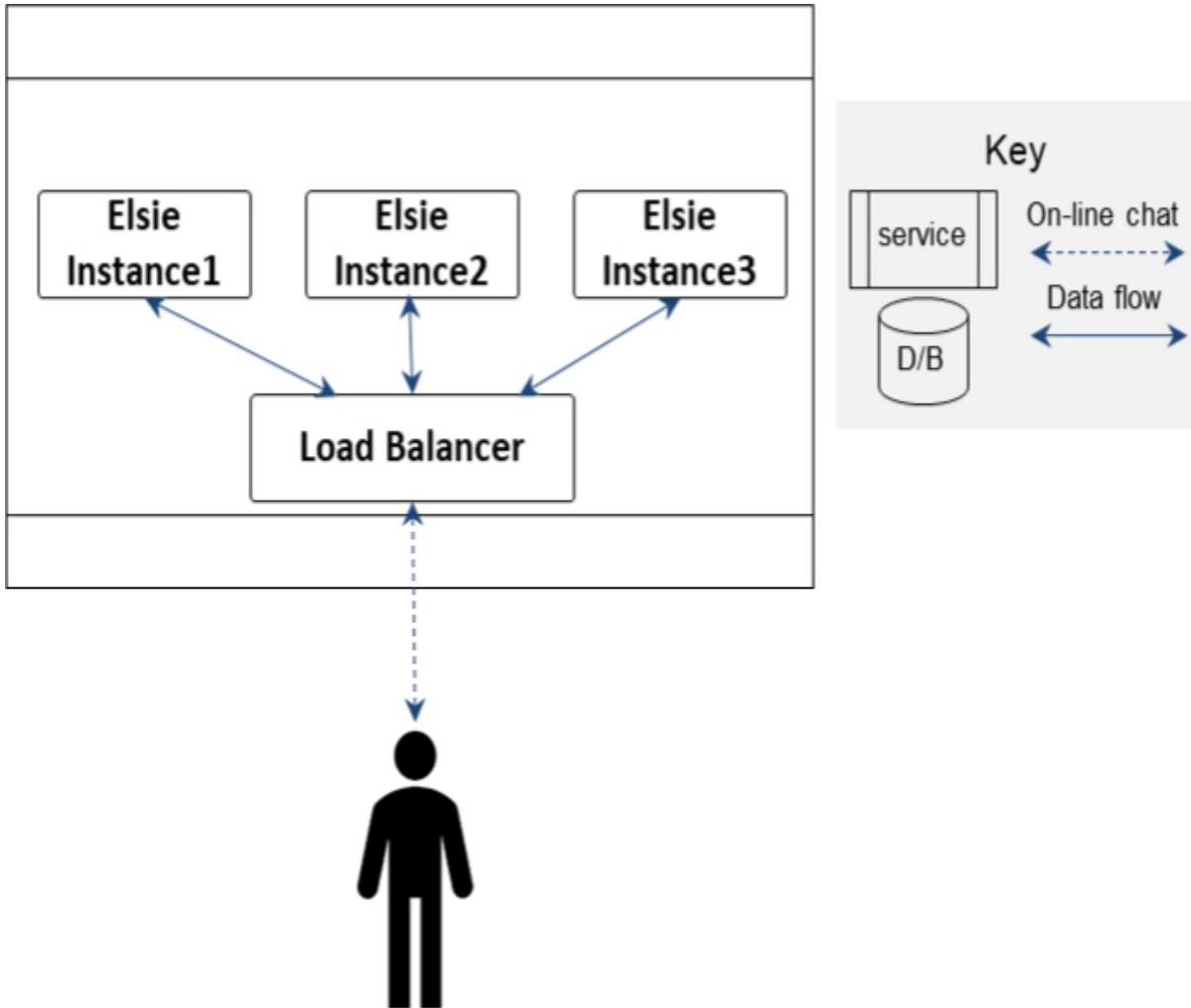
The team’s implementation of Elsie’s natural language interface is successful, and the team initiates performance and scalability testing. Unfortunately, introducing NLU capabilities has a negative impact on some

of the quality attributes. Elsie's performance does not scale up well. The performance goals for Elsie are documented as follows, using the template described earlier in this book:

- *Stimulus*: Elsie can handle interactions with up to 100 concurrent users.
- *Response*: Elsie can process those interactions without any noticeable issues.
- *Measurement*: Response time for any interaction with Elsie is 2 seconds or less at any time.

Testers notice that Elsie's response time exceeds 5 seconds as soon as the number of users exceeds 50, even for simple dialogs. As we saw in [Chapter 5](#), one of the simplest solutions to this issue would be to deploy multiple instances of Elsie and to distribute the user requests as equally as possible between the instances.

For this solution, the TFX team designs the architecture shown [Figure 8.6](#). This architecture uses a load balancer that ensures that user requests are equally distributed between the various instances of Elsie. It also uses a *sticky session* concept, meaning that session data is kept on the same server for the duration of the session in order to minimize session latency.



**Figure 8.6** Implementing multiple instances of Elsie

Those changes are successful. As a result, Elsie can now handle the expected number of queries from importers to business customers' satisfaction

### **Elsie: All-in-One Chatbot**

The team's next challenge is to add to Elsie's capabilities to handle queries from the exporters in addition to those from the importers. They update the models, retrain them, and test them with the business customers.

This approach seems to work well initially, because a single chatbot now handles all dialogs with the TFX users. However, some concerns are emerging. Both the NLU model and the dialog management model are

rapidly growing in size in terms of code and parameters. Training those models now takes several hours even if only small changes are made. Model deployment, including functional, security, and performance testing, also becomes a lengthy process. It is still manageable, but it's soon going to get harder to implement new requests quickly.

Unfortunately, Elsie becomes a bottleneck. The models are becoming harder to maintain and train as they grow. Releasing new versions of the models is getting harder to schedule, as multiple stakeholders need to be satisfied with all the changes to the code and the parameters. Multiple team members are trying to update the models simultaneously, which creates version control issues. The TFX business partners are getting unhappy with the delays in implementing their requests, so the team needs to rethink the architecture of the TFX chatbot.

### ***Elsie: Multidomain Chatbot***

In order to better respond to the requests from their business partners, the team considers using a federated architecture approach for the TFX chatbot. Using this model, the team would create three domain-specific chatbots:

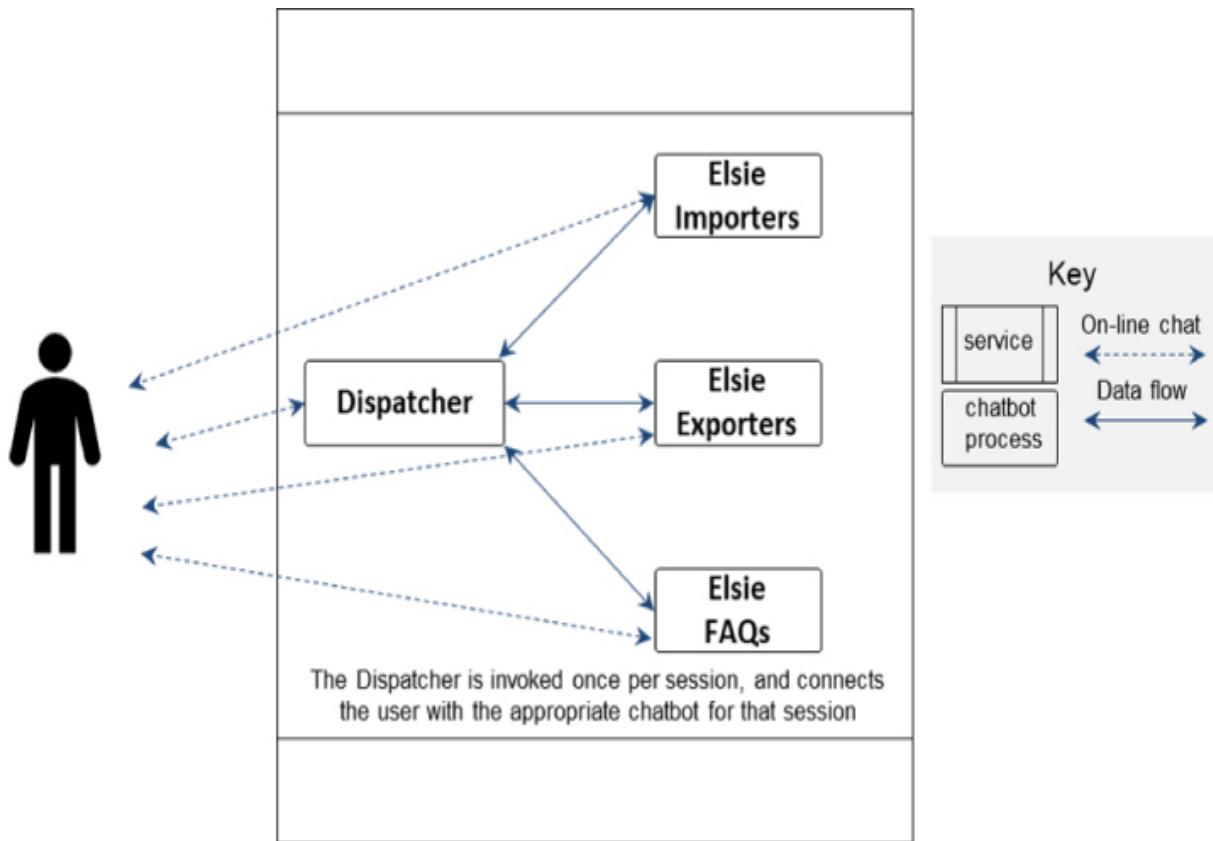
- Elsie for Importers
- Elsie for Exporters
- Elsie for FAQs

The three chatbots would each have its own NLU model and training data. In addition to the three domain-specific chatbots, this architecture would need a routing mechanism to figure out which chatbot should handle a specific user request. One possibility would be to send each request to every chatbot and let it decide whether it can handle the request. Unfortunately, this approach could bring back the performance and scalability issues that the team solved earlier by implementing multiple instances of Elsie. A better strategy would be to implement a dispatcher component (refer to [Figure 8.6](#)) responsible for evaluating each request for a session and deciding which chatbot should handle it. The dispatcher would be implemented as another chatbot that can handle user intents and take action by passing user information to the appropriate domain-specific chatbot.

The team evaluates the tradeoffs between the current and proposed architectures as follows:

- *Maintainability and adaptability versus complexity:* Splitting the all-in-one chatbot models into three smaller model sets would allow the team to respond faster to requests from their business partners. In addition, each chatbot would have its own independent and simpler deployment cycle using the federated architecture model. However, this approach would move complexity around within the TFX chatbot architecture and may increase its fragility. For example, changes made to the open source chatbot framework would have to be tested with and applied to all the chatbots. In addition, multiple sets of training data would need to be managed, which complicates the data ingestion and transformation processes;
- *Performance versus complexity:* Introducing a dispatcher component would improve the throughput of the chatbot but at the cost of increasing complexity as well by introducing a fourth chatbot and associated models into the architecture.

The team feels that the tradeoffs are acceptable and builds a simple prototype to test the proposed approach, including the dispatcher concept. This test is successful, and the team decides to go ahead and implement the architecture depicted in [Figure 8.7](#).



**Figure 8.7** Elsie’s federated architecture

The federated architecture strategy turns out to be successful, and the TFX business customers are pleased with the results.

### ***Benefits of an Architecture-Led Approach***

Chatbots that leverage ML models are data-intensive software systems, and the big data architecture principles and tactics need to be applied to create an effective architecture solution. Similarly to other ML systems, data provenance and metadata are important data architecture considerations for NLU-enabled chatbots.

Some of the architectural tactics discussed in this section are technology specific, and it is important to understand both the issue to be addressed and the architecture capabilities of the chatbot framework being used before deciding to apply a specific tactic. For example, using a federated architecture to solve the all-in-one chatbot challenge discussed earlier may

be unnecessary with some chatbot frameworks, as they partition the complexity and retain the single chatbot concept.

NLU-enabled chatbots are not easy to architect and implement, and some organizations may decide to implement commercial off-the-shelf packages, hoping for a faster and easier implementation. Unfortunately, these packages often introduce new technologies into those organizations. How these technologies fit to the organization's existing architecture is usually not considered, and architects need to come up with strategies to integrate vendor chatbots into their enterprise architecture. In addition to integrating with existing processes and databases, they need to be concerned with how vendor chatbots implement key quality attributes such as security, performance, scalability, and resiliency and whether they meet the organization's requirements standards. We revisit this issue at the end of the next section, which deals with implementing a shared ledger for TFX.

## Using a Shared Ledger for TFX

### Brief Introduction to Shared Ledgers, Blockchain, and Distributed Ledger Technology

A shared ledger is a comprehensive record of transactions, information, or events that are replicated, shared, and synchronized across multiple sites, countries, or institutions. There are a lot of variations among shared ledger technologies. Some of the variations, such as in network governance and network trust, are significant from a business perspective. Some, such as variations in transaction latency and smart contracts, are significant from an architectural perspective. Blockchains and distributed ledger technologies (DLTs) are the most common variations.

A blockchain is generally defined as a secure and immutable list of blocks of information, each of which contains a cryptographic hash<sup>18</sup> of the previous block as well as transaction data.<sup>19</sup> Most people first heard of blockchain as the decentralized data infrastructure behind the digital currency Bitcoin, but blockchains preceded Bitcoin. Bitcoin's innovation was an adaptive proof-of-work task to achieve consensus in a public shared ledger. Startups promoting other digital currencies started imitating this concept, trying to improve on Bitcoin by building new cryptocurrencies on

new and improved blockchains, such as Ethereum.<sup>20</sup> Finally, the idea of a decentralized, cryptographically secure database for uses beyond currency gained adoption, especially in financial services. Trade financing, and especially L/Cs, may be among the most promising use cases, although our TFX case study is architected as a more traditional system.

<sup>18</sup> A *cryptographic hash* is a function that takes an input and returns a unique fixed-length result. The operation is not invertible (you can't re-create the input from the hash).

<sup>19</sup> See Wikipedia, "Blockchain." <https://en.wikipedia.org/wiki/Blockchain>

<sup>20</sup> <https://www.ethereum.org>

A DLT is a shared ledger maintained by multiple parties.<sup>21</sup> It may be thought of as a transaction level-record of consensus with cryptographic verification, whereas a blockchain is a block-level consensus, meaning that the unit of consensus is multiple transactions.<sup>22</sup> The key difference is that a blockchain requires global consensus across all copies of the blockchain to add blocks to the chain, whereas a DLT achieves consensus without having to confirm the addition across the entire chain. DLTs are commonly implemented as private networks but can be implemented as public networks as well (see Appendix B).

<sup>21</sup> *Party* means a person, group, or organization.

<sup>22</sup> Wikipedia, "Distributed Ledger." [https://en.wikipedia.org/wiki/Distributed\\_ledger](https://en.wikipedia.org/wiki/Distributed_ledger)

## Types of Problems Solved by Shared Ledgers, Prerequisites, and Architectural Concerns

Use cases where a shared ledger is the only solution are hard to find. It is relatively easy to identify use cases that can be solved with a blockchain or DLT, but the key question that technologists need to answer when considering using this technology is whether a blockchain- or DLT-based approach is the solution that makes the best tradeoffs. More mature technologies, such as distributed databases, can potentially provide the same or better benefits with a technical implementation that would be easier and less risky. Using an architecture-led approach and specifically assessing

quality attribute tradeoffs is essential when evaluating emerging technologies such as shared ledgers.

Technologists looking for blockchain or DLT use cases often focus on leveraging the benefits of the technology in three areas: security, transparency, and trust. In financial services, prevalent use cases are ones that eliminate the need for both parties to utilize a trusted third party to validate transactions, which increases cost and latency. Some of examples of use cases that are considered promising include the following:

- Clearing and settlement of security trades
- International payments
- Preventing criminals from passing illegally obtained funds as lawful income (anti-money laundering)
- Verifying customer's identity and assessing the appropriateness and risks of their relationships with a financial institution ("know your customer")
- Insurance, including certificates of insurance, reinsurance, and claim processing
- Trade finance, including L/C processing

In addition, some enterprise shared ledgers, such as Corda, are now being considered for adoption by software teams as distributed or decentralized application platforms instead of their immutability or consensus capabilities. A blockchain or a DLT could be a good approach when a complex interfirm or market-level business process needs to be coordinated. This may be a better approach than deploying the associated workflows and business logic on nodes hosted by each participant and having to be intermediated by a new centralized party.

Similarly to what we saw in the AI/ML part of this chapter, lessons from distributed application architectures, and specifically from distributed database architectures, can be applied to shared ledger projects to improve project velocity and reduce risk. Some architectural decisions are likely to be already made for the team. A financial services company may decide to join a consortium to apply this technology, and that decision may be based on business relationships rather than technical reasons. In this scenario, the

decision on which shared ledger technology to use would have already been made by the consortium. Each blockchain technology or DLT is associated with tradeoffs of several quality attribute, such as interoperability, extensibility, performance, scalability, availability, usability, costs, and security. As we mentioned earlier in this chapter, one of the best practices when elaborating an architecture that meets quality attribute requirements is to use a series of appropriate architectural tactics.

Shared ledger implementations in financial services are likely to be using private networks operated by a consortium, at least in the foreseeable future. Although it is technically feasible for a financial services companies to implement its own shared ledger network without joining a consortium, the company would probably not be successful at convincing other firms to join its network. It is also unclear whether financial services firms would ever use public shared ledger networks to transact business. Using a private network is an important constraint for shared ledger architectures in financial services.

A key prerequisite to a private shared ledger network implementation is to have a common interface, accepted by every member within the consortium at a minimum. In practice, this means using industry standard message formats, such as SWIFT (Society for Worldwide Interbank Financial Telecommunication) or ACORD message formats.

## **Shared Ledgers Capabilities**

The traditional public blockchain systems used by cryptocurrencies such as Bitcoin and Ethereum have some limitations. Their security protocols use a consensus-based, probabilistic approach and therefore are compute-intensive and slow at times. To address this issue, two approaches have emerged:

- *Private* shared ledgers that run on servers within a private network operated by one or several companies, usually in the financial services sector. These companies may form a consortium in order to operate their own shared ledger implementation. They may use a DLT or a blockchain;
- *Permissioned* shared ledgers, meaning that the ability to be part of the set of parties who determine which transactions get confirmed is

controlled.

Blockchains and DLTs share some common capabilities. Their data is *immutable* (i.e., it cannot be altered once it is stored in the ledger). However, some DLTs allow rollbacks of transactions in case of an error. Their data is also *verified*, which means that new blocks or transactions are validated and confirmed before being added to the ledger. The validation and confirmation mechanism depends on the specific implementation of a blockchain or a DLT.

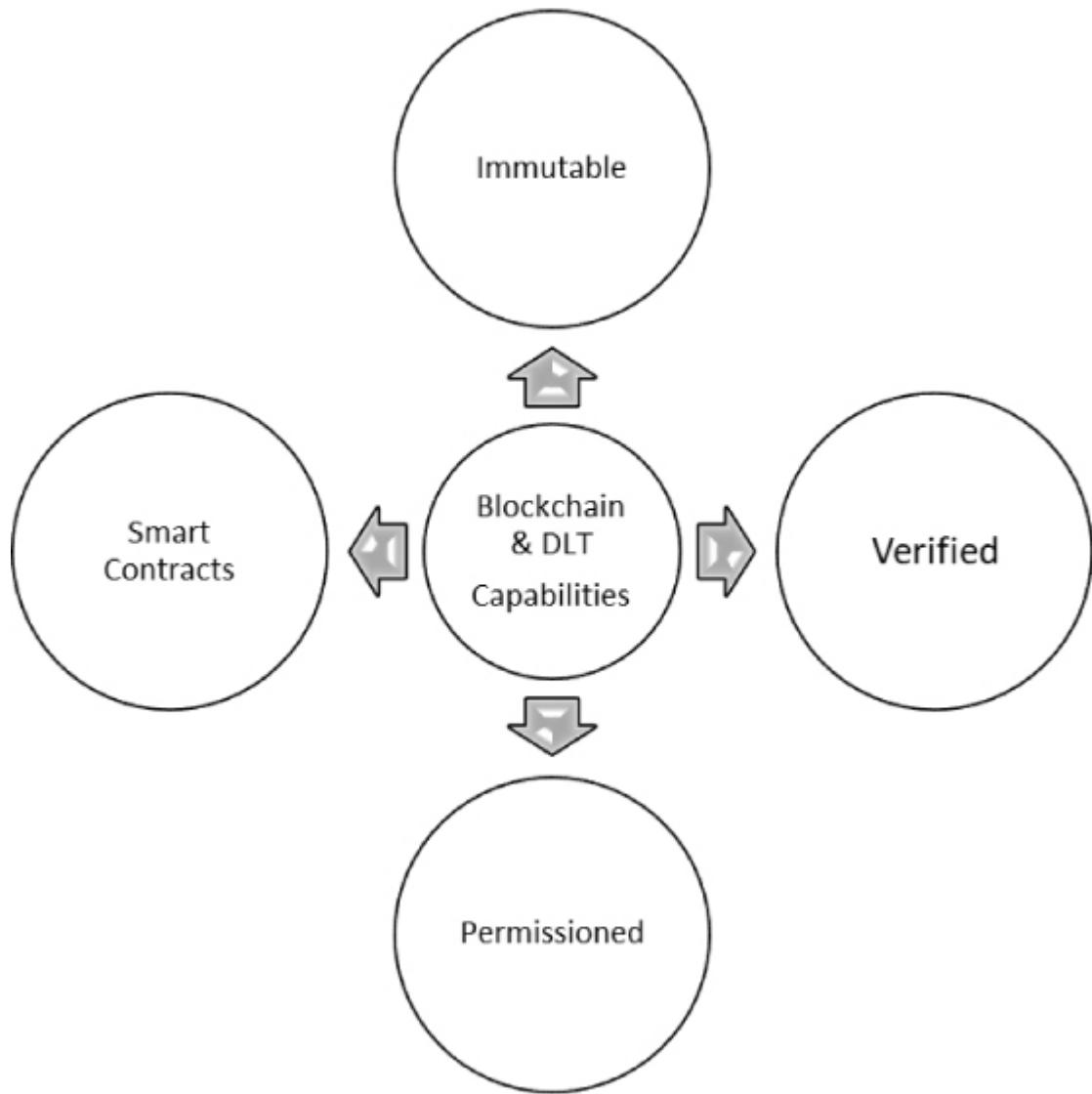
Private blockchains and DLTs are usually *permissioned*, whereas public blockchains and DLTs may or may not be permissioned.<sup>23</sup>

<sup>23</sup> See Richard Gendal Brown, “The Internet Is a Public Permissioned Network. Should Blockchains Be the Same?” *Forbes* (May 6, 2020).

<https://www.forbes.com/sites/richardgendalbrown/2020/05/06/the-internet-is-a-public-permissioned-network-should-blockchains-be-the-same/?sh=44afb8de7326>

Finally, a blockchain or DLT implementation may provide *smart contracts*, which are applications using a set of business rules intended to automatically execute, control, or document events and actions and are triggered by and use ledger data.<sup>24</sup> To use an analogy in the traditional database world, smart contracts can be thought of as sophisticated database triggers. Those capabilities are summarized in [Figure 8.8](#), and [Appendix B](#) provides additional information on specific product implementations.

<sup>24</sup> Wikipedia, “Smart Contract.” [https://en.wikipedia.org/wiki/Smart\\_contract](https://en.wikipedia.org/wiki/Smart_contract)



**Figure 8.8** *Blockchain and DLT capabilities overview*

---

### **When Nontechnical Stakeholders Want to Introduce New Technologies**

Our business partners sometimes want to introduce new technologies, and fit to the organization's architecture usually is not high on their list of selection criteria. They may want to do this for a number of reasons, including because the competition is doing it or because it will make them look innovative.

For example, they may decide to join a shared ledger consortium in order to leverage this emerging technology. Unfortunately, the business benefits of using a shared ledger technology instead of older, more mature technologies such as distributed databases are not always clearly articulated, and implementation challenges are unknown. Also, as we saw earlier in this section, emerging technologies such as ML and chatbots are not easy to architect and implement, and some business organizations may decide to implement commercial off-the-shelf packages, hoping for a faster and easier implementation.

We have found that the best way to deal with this situation is to prevent it from happening in the first place, by developing strong relationships with nontechnical stakeholders and gaining a seat at the table where those decisions are being discussed. Explaining the architectural tradeoffs, including costs, in nontechnical terms may be enough to help people understand that the technology risks may outweigh its benefits.

Should this approach fail and a decision is made to go ahead with the implementation, our next line of defense would be to get a commitment to replace the new technology with one that fits the organization's architecture in the medium to long term (3 to 5 years). The short-term solution should be validated for compliance with the quality attribute requirements as well as for compliance with the standards of the organization before implementation.

If possible, the new technology should be segregated from other systems, using a set of well-defined APIs, potentially managed by an API gateway. Like any other system within the organization, the new technology should be integrated into the organization's monitoring and alerting systems.

## **Implementing a Shared Ledger for TFX**

Going back to the TFX case study, the team assumes that the company would like to explore whether the capabilities of a shared ledger could be applied for the TFX platform. Could a shared ledger be used to automate some of the L/C processes by eliminating intermediaries and removing friction?

The company decides to join a consortium whose members already include several banks, one of which is the first client of the TFX platform. This consortium operates a DLT platform over a private network and does not currently have any L/C application software running on its platform. It is interested in using its DLT network for trade financing and would like to implement an L/C application on that network. It asks our software company to develop this new software, called TFX-DLT, as an addition to the TFX system. It defines the initial business objectives for a pilot project using TFX-DLT as follows:

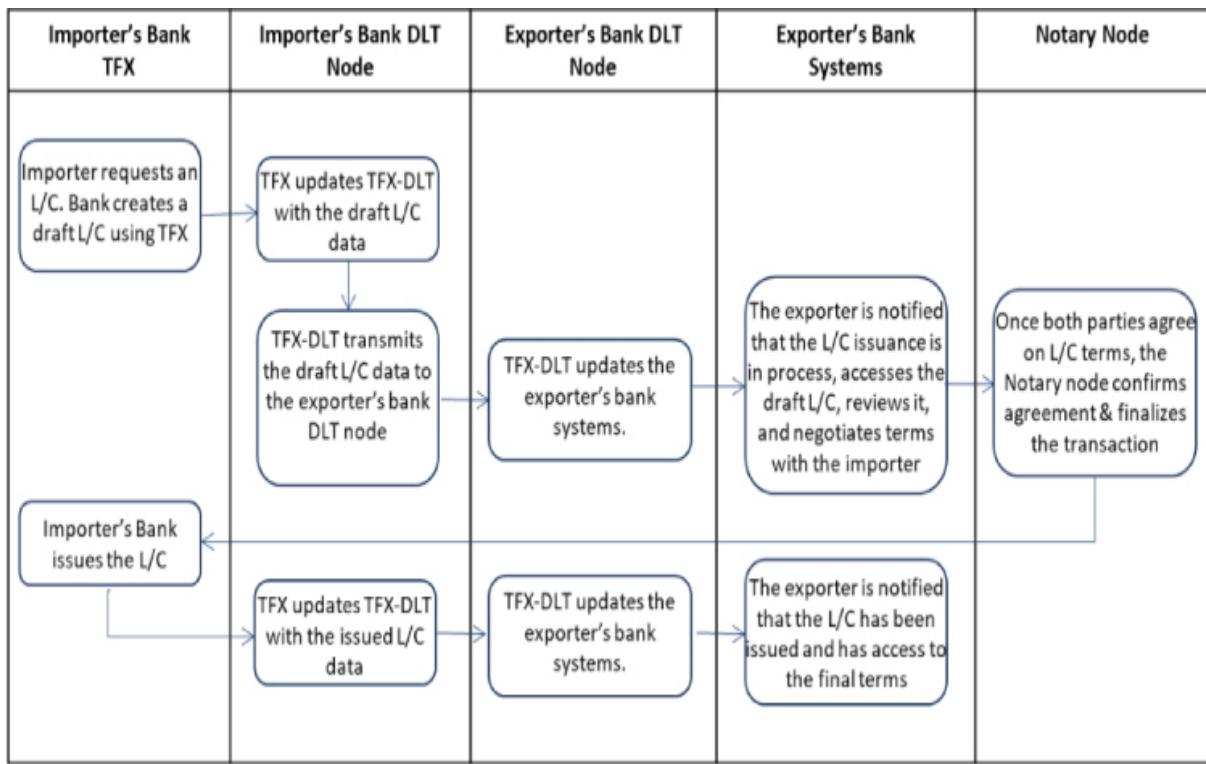
- To improve the L/C issuance and amendment processes by providing tools that enable the importer and the exporter to record L/C contract and amendment terms and immediately share them with all interested parties during negotiation.
- To improve the L/C payment processes by providing capabilities to confirm and notify all parties that payment has been made.

### ***L/C Issuance Using a DLT***

As mentioned at the beginning of this chapter, when dealing with emerging technologies, it is important to do hands-on prototyping to gain knowledge about that technology before proceeding to implementation. The team's first task is to get familiar with the DLT used by the consortium and to write a simple smart contract application using that DLT. Once this first task is complete and the team has gained some experience with the DLT, they start designing the new L/C issuance flow and architecting the TFX-DLT application. The team makes the following assumptions; see [Figure 8.9](#) for a depiction of the new L/C issuance flow:

- The importer's bank uses the TFX platform. Ideally, the first implementation of TFX-DLT on the importer's side would be at the client banking company.
- Both the importer's bank and the exporter's bank are part of the consortium and are running DLT network nodes.
- The exporter's bank systems can be interfaced to the TFX-DLT application using a set of custom APIs to be built by the team as part of the TFX-DLT project.

The team architects TFX-DLT based on the new issuance flow, as depicted in [Figure 8.9](#). In this flow, the notary node confirms the L/C issuance transaction once both parties have agreed on terms. Interim changes to the L/C terms occurring during the negotiation are not recorded on the DLT; only the initial and final terms are recorded. Using TFX-DLT does not eliminate the need for the importer and exporter to communicate directly with each other while they negotiate the terms of the L/C. However, the L/C terms need to be recorded on TFX-DLT once they have been finalized in order to be confirmed by the notary node, which is part of the core DLT platform.

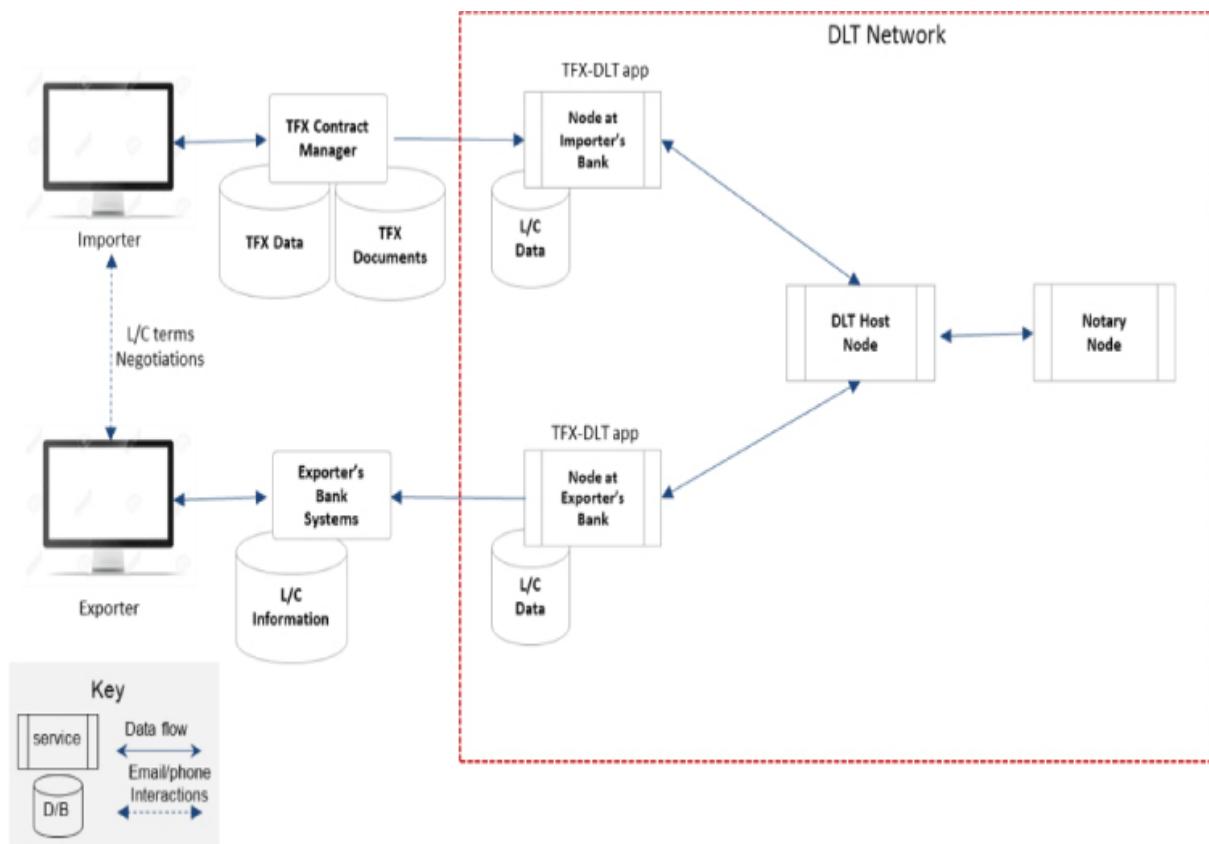


**Figure 8.9 TFX-DLT L/C issuance flow**

The next step is to design a system architecture to enable this flow. That architecture is depicted in [Figure 8.10](#). The team also designs a simple set of APIs to interface both TFX (specifically, the Contract Manager Service) and the exporter's bank systems to TFX-DLT. As shown on the TFX-DLT architecture diagram ([Figure 8.10](#)), L/C data is stored on TFX, on the DLT nodes, and in the exporter's bank systems. The team considers two options for storing the L/C documents:

- Similarly to the approach for L/C data, the first option is to store them on the DLT as well as on TFX and on the exporter's bank systems.
- The second option is to store a URL on the DLT. This URL points to a location within the TFX platform and allows the exporter's bank to access the L/C documents.

The team looks at the tradeoffs between each option and decides to use the second option because the size of the documents may cause performance issues with the DLT. In addition, because the documents reflect exactly the L/C terms stored on TFX-DLT, which are immutable, storing the L/C issuance documents outside TFX-DLT does not create any trust issues. The team also decides to store a hash code for each document in the DLT as part of the L/C data in order to identify any changes.



**Figure 8.10 TFX-DLT architecture overview—L/C issuance and amendments**

The team is able to implement this architecture in a fairly short period of time. They conduct a pilot experiment with their major client and several

other consortium members that have implemented a DLT node and have agreed to try using TFX-DLT. After a few months, the new platform demonstrates the following benefits:

- Banks are satisfied with the new process, as it allows them to issue and advise L/Cs with less manpower and less rework because all parties have access to the same information during the L/C issuance process.
- In addition, the parties involved in this process can trust that they are accessing the same data on both the importer's and the exporter's sides and that the data cannot be altered accidentally or otherwise. As a result, importers and exporters can agree on L/C terms more rapidly and without errors.

The team is also able to extend the architecture to handle the L/C amendment process and implement this capability successfully, with similar benefits to those provided by TFX-DLT for the L/C issuance process.

### ***L/C Payment Using a DLT***

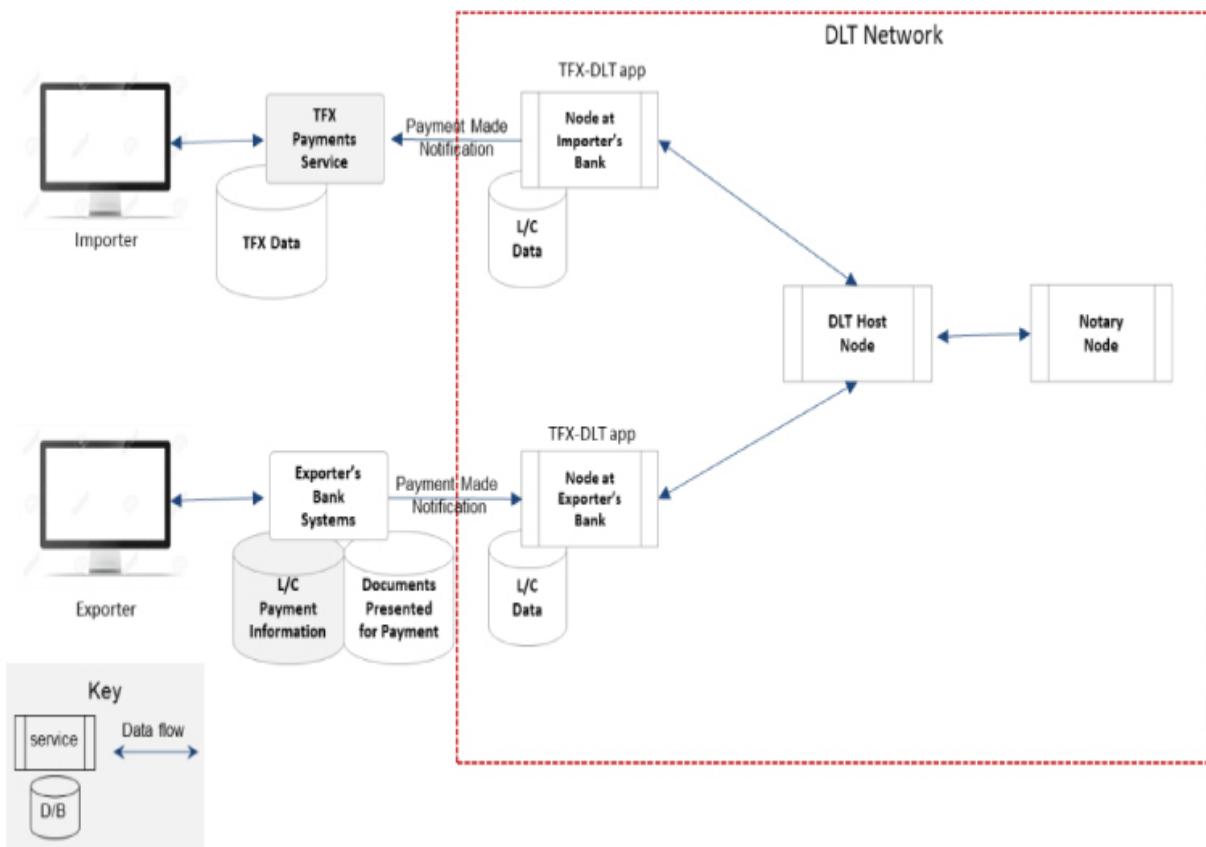
The team now tackles their second objective: improving the L/C payment process. The L/C data is already stored on TFX-DLT, so the team implements the following capabilities:

- A new API to be used by the exporter's bank to notify TFX-DLT that a payment has been made for a specific L/C.
- An interface between TFX-DLT and the TFX Payments Service, using the TFX Payments Service API. This interface is used to notify the TFX Payments Service that a payment has been made by the exporter's bank for a specific L/C.
- Similarly to the approach used for L/C issuance documents, documents presented for payment are not stored on the DLT. Instead, they are stored in the exporter's bank system, and a corresponding URL is stored on the DLT. This URL points to a location within the exporter's bank system and allows access to the documents by all parties.

The team is able to rapidly extend the TFX-DLT architecture to include those capabilities (see [Figure 8.11](#)). They conduct a second pilot project with their major client and several other consortium members that are

already using TFX-DLT for issuing and amending L/Cs and have agreed to use it for L/C payments as well. At the end of the project, the new platform demonstrates the following benefits:

- Banks are satisfied with the new process, as the importer's bank is notified that an L/C has been paid as soon as the payment has been made to the exporter.
- When partial shipments are allowed under the terms of the L/C, the importer's bank is able to easily reconcile the payments made for each shipment against the total L/C amount and quickly identify any discrepancies.
- Both the exporter and the importer are satisfied with the new process, as it enables a quicker and simpler payment process with fewer issues.
- As for the L/C issuance and amendment processes, all the parties involved in the payment process can trust that they are accessing the same payment data on both the importer's and the exporter's sides and that it is immutable.



## **Figure 8.11: TFX-DLT architecture overview—L/C payments**

In summary, TFX-DLT enables the major client to efficiently and securely issue and pay L/Cs using the TFX system when dealing with counterparty banks that are part of the DLT consortium and do not use the TFX system.

### **Benefits of an Architecture-Led Approach**

Shared ledger implementations have significant architectural challenges, such as latency, throughput, and access to external information. Teams working on those projects often focus on programming challenges associated with the platform and may ignore these other challenges because no one is thinking about an overall architecture. Using an architecture-led approach addresses this problem.

Some architectural decisions are sometimes made for you when a platform is selected. Similarly to NoSQL products,<sup>25</sup> each shared ledger technology is associated with several quality attribute tradeoffs, including interoperability, extensibility, performance, scalability, availability, usability, costs, and security. Each platform considered for use in the organization should go through a validation of architectural characteristics with focus on quality attributes and how well the platform meets the standards of the organization. This review helps the organization understand the impact of the shared ledger technology on its systems. For example, data security on the DLT network would be handled by the DLT and would be beyond the team's control. However, the TFX-DLT architecture would still need to secure the interface between the bank's servers and the DLT server located at the bank.

<sup>25</sup> Ian Gorton and John Klein, “Distribution, Data, Deployment: Software Architecture Convergence in Big Data Systems,” *IEEE Software* 32, no. 3 (2014): 78–85.

One of the best practices when elaborating an architecture that meets quality attribute requirements is to use a series of appropriate architectural tactics.<sup>26</sup> As mentioned earlier, tactics are often documented in catalogs to promote reuse of this knowledge among architects. A few existing catalogs contain tactics specific to blockchain and DLT systems,<sup>27</sup> and catalogs of tactics for distributed software architectures can be used as well.<sup>28</sup>

<sup>26</sup> Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice*, 3rd ed. (Addison-Wesley, 2012).

<sup>27</sup> For example, see Raghvinder S. Sangwan, Mohamad Kassab, and Christopher Capitolo, “Architectural Considerations for Blockchain Based Systems for Financial Transactions,” *Procedia Computer Science* 168 (2020): 265–271.

<sup>28</sup> For a catalog of software architecture tactics for big data systems, see [https://quabase.sei.cmu.edu/mediawiki/index.php/Main\\_Page](https://quabase.sei.cmu.edu/mediawiki/index.php/Main_Page)

Establishing standards and ensuring that they are globally adopted may be the most important factor in the long-term adoption of shared ledger platforms. Consortiums help establish common standards, reference architectures, and common practices among their members, but interoperability between networks operated by consortiums as well as with other shared ledger networks is a challenge. There are potentially excellent use cases in trade financing; however, we may be still years away from mainstream adoption. For a shared ledger based solution to work effectively, every participant involved in the Letter of Credit process would need to run a shared ledger network node. To keep this in perspective, SWIFT was founded in 1973 by 239 banks in 15 countries and, by 2010, was adopted by more than 11,000 financial institutions in 200 countries. SWIFT is a success story, but global adoption didn’t happen overnight.

Shared ledger networks may end up being positioned in the same manner as the SWIFT network: a universal but fairly obscure “plumbing” framework understood by a few specialists while everyone else hides that framework behind an API or data format.

## Summary

In this chapter, we showed the value of an architecture-driven approach when dealing with emerging technologies. We presented a brief overview of three key emerging technologies: artificial intelligence, machine learning, and deep learning. We discussed how an architecture-led approach can be used to implement those technologies within the context of the TFX case study, specifically for document classification and for implementing a customer service chatbot. We then turned our attention to another key

emerging technology, shared ledgers, and explained how an architecture team such as the TFX team would support its implementation.

We discussed the use of those technologies in practical terms in the context of the TFX case study. Specifically, we discussed the types of problems that can be solved by ML, its prerequisites, and its architecture concerns. We described how ML can be used for document classification in order to automate part of the document verification process in the TFX platform. We gave a brief overview of the various components of an ML architecture and process from an architect's perspective. We highlighted the need to architect data when working with ML and the applicability of lessons learned when architecting big data and advanced analytics systems. We also pointed out the need to consider architecture quality attributes such as security, performance, and scalability when designing an application that relies on ML and the benefits of relying on architecture tactics when designing an architecture that meets these requirements.

We then described how a chatbot could be built incrementally for the TFX platform to provide information to both importers and exporters with minimal human intervention from L/C specialists. As part of this narrative, we looked at the importance of taking an architecture-first approach and especially considering architectural quality attributes such as security and performance up front in the project.

We finally gave a brief overview of shared ledgers and described how a shared ledger network could be implemented for TFX. We looked at two potential use cases, including L/C issuance and amendments, as well as L/C payments.

In summary, using concrete examples from our case study, we illustrated how architecture plays a key role in applying emerging technologies and adopting them across the enterprise. We took a deeper look at a few of those technologies, but others will doubtless emerge in the near future. The key takeaway is that an architecture-led approach enables the smooth adoption of innovative technologies across the enterprise and significantly decreases implementation and technology risks. This approach basically includes applying the Continuous Architecture principles, specifically principle 2, *Focus on quality attributes, not functional requirements*, principle 4,

*Architect for change—leverage the “power of small,” and principle 5, Architect for build, test, deploy, and operate.*

## Further Reading

This section includes a list of books and websites that we have found helpful to further our knowledge in the topics discussed in this chapter.

- For a classic textbook on AI, Stuart Russell and Peter Norvig’s *Artificial Intelligence: A Modern Approach*, 4th ed. (Pearson, 2020) is a great resource. It has been called “the most popular artificial intelligence textbook in the world”<sup>29</sup> and is considered the standard text in the field of AI, including ML.
- *The Hundred-Page Machine Learning Book* (Andriy Burkov, 2019) by Andriy Burkov is a great condensed textbook on ML. This book provides accurate and helpful information on most popular ML techniques. It is not as exhaustive as *Artificial Intelligence: A Modern Approach* but is still very useful for most of us.
- Charu C. Aggarwal’s *Neural Networks and Deep Learning* (Springer, 2018) is a good textbook for readers who want to better understand neural networks and deep learning concepts.
- *Distributed Ledger Technology: Beyond Block Chain: A Report by the UK Government Chief Scientific Adviser* (Government Office for Science, 2016)<sup>30</sup> is a comprehensive and objective analysis of DLTs (as of end 2015). It also includes a useful set of references (see pages 84–86).
- Raghvinder S. Sangwan, Mohamad Kassab, and Christopher Capitolo’s “Architectural Considerations for Blockchain Based Systems for Financial Transactions,” *Procedia Computer Science* 168 (2020): 265–271, contains some architectural tactics applicable to DLTs.

<sup>29</sup> Kevin Gold, “Norvig vs. Chomsky and the Fight for the Future of AI” [blog post], Tor Books Blog (June 21, 2011). <https://www.tor.com/2011/06/21/norvig-vs-chomsky-and-the-fight-for-the-future-of-ai>

<sup>30</sup>

[https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment\\_data/file/49](https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/49)

[2972/gs-16-1-distributed-ledger-technology.pdf](#)

# Chapter 9. Conclusion

*Make the work interesting and the discipline will take care of itself.*

—E. B. White

## What Changed and What Remained the Same?

As stated in the introduction, the foundations of software architecture haven't changed in the last few years. The overall goal of architecture is still to enable early and continual delivery of business value from the software being developed. Unfortunately, this goal isn't always prioritized or even well understood by many architecture practitioners.

The components of a software architecture do not exist in isolation: they are interrelated. Creating an architecture means making a series of tradeoffs between requirements, decisions, blueprints, and technical debt items, which are always reflected into the executable code. Sometimes those tradeoffs are optimal, and sometimes they are the least unfavorable ones because of constraints beyond the team's control. Either way, consciously making architectural tradeoffs and explicitly focusing on architectural decisions is an essential activity.

While our first book on Continuous Architecture was more concerned with outlining and discussing concepts, ideas, and tools, the book you just read provided more hands-on advice. It focuses on giving guidance on how to leverage the continuous architecture approach and includes in-depth and up-to-date information on major architecture concerns including data, security, performance, scalability, resilience, and emerging technologies.

With this book, we revisited the role of architecture in the age of agile, DevOps, DevSecOps, cloud, and cloud-centric platforms. Our goal was to provide technologists with a practical guide on how to update classical software architecture practice in order to meet the complex challenges of today's applications. We also revisited some of the core topics of software architecture, such as the evolving role of the architect in the development

team, meeting stakeholders' quality attribute needs, and the importance of architecture in managing data and in achieving key cross-cutting concerns. For each of these areas, we provided an updated approach to making the architectural practice relevant, often building on conventional advice found in the previous generation of software architecture books and explaining how to meet the challenges of these areas in a modern software development context.

---

## Is Architecture Still Valuable?

What is the real value of architecture? We think of architecture as an enabler for the delivery of valuable software. Software architecture's concerns, including quality attribute requirements, are at the heart of what makes software successful.

A comparison to building architecture may help illustrate this concept. Stone arches are one of the most successful building architecture constructs. Numerous bridges with stone arches, built by the Romans around 2,000 years ago, are still standing—for example, the Pont du Gard was built in the first century AD. How were stone arches being built at that time? A wooden frame known as a *centring* was first constructed in the shape of an arch. The stonework was built up around the frame, and finally a keystone was set in position. The keystone gave the arch strength and rigidity. The wooden frame was then removed, and the arch was left in position. The same technique was used in 1825–1828 for building the Over Bridge in Gloucester, England (see [Figure 9.1](#)).

A large, light gray rectangular box with a thin black border, centered on the page. Inside the box, the word "placeholder" is printed in a bold, black, sans-serif font.

**placeholder**

## **Figure 9.1 Centring of Over Bridge, Gloucester, England**

We think of software architecture as the centring for building successful software “arches.” When Romans built bridges using this technique, it is likely that none worried about the aesthetics or the appearance of the centring. Its purpose was the delivery of a robust, strong, reliable, usable, and long-lasting bridge.

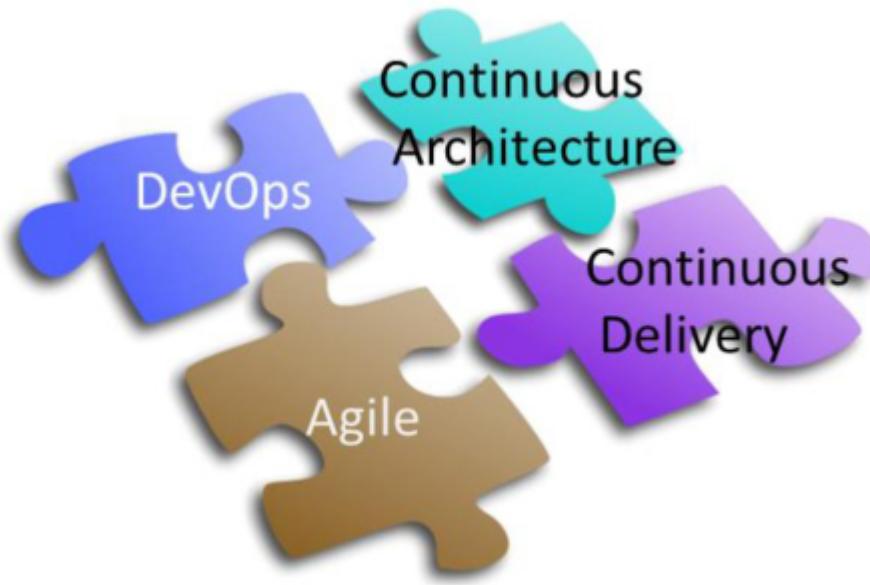
Similarly, we believe that the value of software architecture should be measured by the success of the software it is helping to deliver, not by the quality of its artifacts. Sometimes architects use the term *value-evident architecture* to describe a set of software architecture documents they created and are especially proud of and that developers should not (ideally) need to be sold on in order to use the architecture. However, we are somewhat skeptical about these claims—can you really evaluate a centring until the arch is complete, the key stone has been put in place and the bridge can be used safely?

## **Updating Architecture Practice**

As Kurt Bittner noted in the preface to our first book, *Continuous Architecture*,<sup>1</sup> Agile development has become old news, and older iterative, incremental, and waterfall ways of developing software have faded from the leading edge of new software development. As older applications are replaced, refactored, or retired, older methodologies are losing their grip on software developers. In the meantime, agile is evolving thanks to the concepts of DevOps, DevSecOps, and continuous delivery. And software architecture is catching up by adopting modern approaches such as Continuous Architecture and evolutionary architecture.<sup>2</sup> Each of those approaches, processes, and methodologies can be thought of as pieces of a software delivery puzzle (see Figure 9.2).

<sup>1</sup> Murat Erder and Pierre Pureur, *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World* (Morgan Kaufmann, 2015).

<sup>2</sup> Neal Ford, Rebecca Parsons, and Patrick Kau, *Building Evolutionary Architectures* (O’Reilly Media, 2017).



**Figure 9.2** *The software delivery puzzle*

One of the main reasons the Continuous Architecture approach works well is that it is not a formal methodology. There is no preset order or process to follow for using the Continuous Architecture principles, essential activities, techniques, and ideas; and you may choose to use only some of them depending on the context of the software product you are working on. We have found that using this toolbox is very effective for our projects, and the Continuous Architecture tools are dynamic, adaptable, and evolve over time. These tools were built and fine-tuned from our experience with real projects and our work with other practitioners. They are practical, not theoretical or academic. They enable the architect to help software development teams create successful software products that implement key quality attributes.

Chapter 2, “Architecture in Practice: Essential Activities,” does not emphasize models, perspectives, views, and other architecture artifacts. Those are valuable deliverables that should be leveraged to describe and communicate the architecture. However, architecture artifacts on their own are insufficient without performing the following essential activities to build and maintain an architecture:

- Focusing on quality attributes, which represent the key cross-cutting requirements that a good architecture should address. We believe that

security, scalability, performance, and resilience may be becoming the most important quality attributes for a modern architecture.

- Driving architectural decisions, which are the unit of work of architectural work.
- Knowing your technical debt, the understanding and management of which is key for a long-lasting architecture.
- Implementing feedback loops, which enable us to iterate through the software development life cycle and understand the impact of architectural decisions in an efficient manner.

## Data

An important development in the field of architecture during the last few years has been an increased focus on data. Data is an extremely fast-evolving and interesting architectural topic.

Processing data is the reason that IT systems exist. The goal of almost every technology that has been developed to date is to make sure that data is processed more effectively and efficiently. With the proliferation of connected devices and Internet usage, the amount of data in the world is increasing exponentially. This has created a need to manage, correlate, and analyze data in flexible ways so that we can find connections and drive insights. The technology response has had a significant impact on how systems are developed. From a data architectural perspective, we identified three main drivers:

- Explosion of different database technologies emerging initially from the Internet giants, known as FAANG (Facebook, Amazon, Apple, Netflix, and Google).
- Ability of data science to leverage cheap and scalable technology for corporations to drive strategic value from the large amount of data being generated in an increasingly digital world.
- Focus on data from a business and governance perspective. This is particularly true for highly regulated industries such as pharmaceuticals and financial services.

Before these factors came to the fore, data architecture was the domain of a small set of data modelers and technologists who dealt with reporting and used technologies such as data warehouses and data marts. Many systems were designed from a purely functional perspective with the design of the system's data being something of an afterthought. But in today's world of increasingly distributed systems, the focus on data needs to become a central architectural concern. Here's some advice on how to do this:

- When focusing on data, the first aspect to consider is how you communicate. Developing a common language between the team and the business stakeholders enables an effective and efficient software development life cycle. Domain-Driven Design is one recommended approach that embraces common language as well as enables development of loosely coupled services. Most important, the common language should live within your code and other artifacts.
- Database technology is a very vibrant market offering a wide variety of options to choose from. A choice of database technology is a significant decision, because changing it will be an expensive task. Base your selection on the quality attribute requirements.
- Finally, data is not an island—it needs to be integrated between components and with other systems. When you think about data, do not focus only on the technology and how to model it; think about how it will integrate. Focus on being clear about which components master the data and how you define (i.e., represent and identify) the data and associated metadata. Finally, be clear on how to manage the evolution of the definitions over time.

## Key Quality Attributes

We believe that a set of specific quality attributes are becoming more important for architectures in the age of DevOps, DevSecOps, and cloud. Those quality attributes are not always well understood or prioritized by software architects and engineers, and this is the reason we discussed them in detail in this book.

Let us use a real-life example to illustrate the importance of these quality attributes. The year 2020 was one of dramatic volatility for the stock

market. As concerns about the impact of COVID-19 sent the market plunging and then monetary policy and the promise of a vaccine sent them soaring back up, investors and traders scrambled to put in trades to limit loss or in hopes of profit. But on the days when the market was the most volatile, the days when there was the most to lose or gain, the days when activity was the highest, these people found that they were unable to enter trades or even see their portfolios online. Some were unable to log in, some who logged in were unable to enter trades in a timely manner, and some found that their accounts were showing up with an erroneous zero balance. As events with similar negative effects may occur in the future for all types of systems, we recommend focusing on those key quality attributes when architecting new systems or enhancing existing ones.

Overall, it isn't possible for us to give you a single architectural pattern or recipe to make your system secure, scalable, performant, and resilient. Creating a system involves many design decisions at different levels of abstraction that, when combined, allow the system to meet all of its quality attribute requirements. Nevertheless, we have included some advice on each key quality attribute in this section. We don't claim that this list is comprehensive, but it is a toolset that we have found to be generally useful.

## Security

Today, security is everyone's concern, a critical one for every system given the threats that modern Internet-connected systems face and the stringent security-related regulations, such as the General Data Protection Regulation privacy regulations, that we all need to comply with. We offer the following advice for making your systems more secure:

- Integrate security into your architecture work as a routine part of the Continuous Architecture cycle, using proven techniques from the security community, such as threat modeling to identify and understand threats. You need to do this before you attempt to mitigate security threats by using security mechanisms. In addition, you need to focus on improving processes and on training people.
- Security work also has to adapt to the agile way of delivery, given the need today to move quickly and continually while the same time addressing the increasing sophistication of the threat landscape. It is no

longer useful or effective to schedule multiday security reviews twice a year and produce pages and pages of findings for manual remediation. Today, security needs to be a continual, risk-driven process that is automated as much as possible and supports the continuous delivery process rather than hindering it. By working in this cooperative, risk-driven, continuous way, you can ensure that your systems are ready and continue to be ready to face the security threats that they are inevitably going to face.

## Scalability

Scalability has not been traditionally at the top of the list of quality attributes for an architecture, but this has changed during the last few years, as we have seen with the trading outages mentioned earlier in this section. Here are some recommendations on making your systems more scalable:

- Do not confuse scalability with performance. Scalability is the property of a system to handle an increased (or decreased) workload by increasing (or decreasing) the cost of the system. Performance is “about time and the software system’s ability to meet its timing requirements.”<sup>3</sup>
- Like other quality attributes, a complete perspective for scalability requires you to look at business, social, and process considerations as well as its architectural and technical aspects.
- You can use a number of proven architectural tactics to ensure that your systems are scalable, and we have presented some of the key ones in this book.
- Keep in mind that performance, scalability, resilience, usability, and cost are tightly related. We recommend that you use scenarios to document scalability requirements and help you create scalable software systems.
- Remember that for cloud-based systems, scalability isn’t the problem of the cloud provider. Software systems need to be architected to be scalable, and porting a system with scalability issues to a commercial cloud will probably not solve those issues.

- Include mechanisms in your systems to monitor their scalability and deal with failure as quickly as possible. These mechanisms will enable you to better understand the scalability profile of each component and quickly identify the source of potential scalability issues.

<sup>3</sup> Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice*, 3rd ed. (Addison-Wesley, 2012).

## Performance

Unlike scalability, performance has traditionally been at the top of the list of quality attributes for an architecture because poor performance may have a serious impact on the usability of your system. We offer the following advice for dealing with performance challenges:

- As for scalability, there are a number of proven architectural tactics that you can use to ensure that modern systems provide acceptable performance, and we have presented some of the key ones in this book.
- Use scenarios to document performance requirements.
- Design your systems around performance modeling and testing. In a continuous delivery model, performance tests are an integral part of the software system deployment pipeline.
- Include mechanisms in your systems to monitor their performance and to deal with failure.

## Resilience

In recent years, a new approach to achieving availability and reliability has emerged, which still uses many of the fundamental techniques of data replication and clustering but harnesses them in a different way. Rather than externalizing availability from systems and pushing it into specialist hardware or software, architects today are designing systems to recognize and deal with faults throughout their architecture. Following decades of engineering practice in other disciplines, we term these systems *resilient*. We offer the following suggestions to make your systems resilient:

- When architecting a resilient system, you need to include mechanisms to quickly recognize when problems occur, isolate problems to limit their impact, protect system components that have problems to allow recovery to take place, mitigate problems when system components cannot recover themselves, and resolve problems that cannot be quickly mitigated.
- Use scenarios to explore and document resilience requirements.
- A number of proven architectural tactics can be applied to create resilience mechanisms your systems, and we have discussed several of the more important ones in this book. However, remember that achieving resilience requires focus on the entire operational environment—people and processes as well as technology—in order to be effective.
- Implement a culture of measuring and learning to learn from your successes and failures, continually improve the level of resilience of your systems, and share your hard-won knowledge with others.

## The Architect in the Modern Era

As mentioned in [Chapter 2](#), architecture activities have become a team responsibility in the agile, DevOps, and DevSecOps era. Architecture is increasingly evolving into a discipline, or skill, rather than a role. The separation between software architecture and software engineering appears to be vanishing. However, we can highlight the key skills required for conducting architectural activities as follows:

- *Ability to design*: Architecture is a design-oriented activity. An architect might design something quite concrete, such as a network, or something less tangible, such as a process, but design is core to the activity.
- *Leadership*: Architects are not just technical experts in their areas of specialization—they’re technical leaders who shape and direct the technical work in their spheres of influence.
- *Stakeholder focus*: Architecture is inherently about serving a wide constituency of stakeholders, balancing their needs, communicating

clearly, clarifying poorly defined problems, and identifying risks and opportunities.

- *Ability to conceptualize and address systemwide concerns:* Architects are concerned about an entire system (or system of systems), not just one part of it, so they tend to focus on systemic qualities rather than detailed functions.
- *Lifecycle involvement:* An architect might be involved in all phases of a system's life cycle, not just building it. Architectural involvement often spans a system's entire life cycle, from establishing the need for the system to its eventual decommissioning and replacement.
- *Ability to balance concerns:* Finally, across all these aspects of the job, there is rarely one right answer in architectural work.

DevOps, DevSecOps, and the shift left movement have blurred many traditional roles, including architecture, but the responsibilities and tasks associated with the traditional roles still need to be done. Roles such as performance engineering, security engineering, and usability engineering fall into this category. In each case, there is a lot of work to be done outside of architecture and implementation; however, if the architecture is not suitable, then specialized tooling and processes can't rescue the system.

## Putting Continuous Architecture in Practice

If you read this entire book, you should feel comfortable putting the Continuous Architecture principles and tools in practice. We hope that even if you only looked into sections that you find relevant, you found useful information and tactics. Success for architecture is the ability to influence what we called the “realized architecture” (i.e., the software product implemented as code on a particular environment). Similarly, success for us would be for practitioners to put Continuous Architecture recommendations immediately into practice.

We believe that the current trend in the industry is away from traditional architecture methodologies, and we do not expect the pendulum to swing back. There is a need for an architectural approach that can encompass modern software delivery methods, providing them with a broader

architectural perspective, and this is exactly what Continuous Architecture is about.

We hope that you have found this book interesting and useful and that you will be inspired by it to adapt and extend its advice with new ideas on how to provide architecture support to your projects to deliver robust and effective software capabilities rapidly. Good luck, and share your experiences with us.

# Appendix A. Case Study

*Few things are harder to put up with than the annoyance of a good example.*

—Mark Twain

This appendix describes the case study used throughout the book to illustrate the architectural techniques introduced in each chapter. Of course, the tradeoff is that it is difficult to create a case study that clearly illustrates each point we want to make without the case study becoming excessively complicated and rather unrealistic. However, we feel that we have managed to largely resolve this tension satisfactorily.

We chose for the case study a cloud-centric application in the trade finance domain, as we feel that it probably has enough similarities to systems you are likely to have worked on to be quickly understood but is complex enough to illustrate the chapter topics. The application is designed to have enough architecturally interesting features that we can illustrate our points without being artificial.

Trade finance is an area of global financial business that underpins global trade in physical goods. The idea of trade finance is to use a range of financial instruments to reduce the risk of sellers of goods not being paid and receivers of goods paying for but not receiving goods. It involves financial intermediaries (typically banks) providing guarantees to buyers and sellers that the transaction will be underwritten by the financial institution given certain conditions.

We cannot fully describe a realistic trade finance system, so we focus our case study on one specific part of trade finance—letters of credit—and provided an outline architectural description, parts of which we fill in as we use the case study in the main chapters of the book. In [Chapter 1, “Why Software Architecture Is More Important than Ever,”](#) we provide a high-level overview of the business domain and the key actors and use cases of the system. This appendix expands that description sufficiently to be used as a foundation for the rest of the book.

# Introducing TFX

A letter of credit (L/C) is a commonly used trade finance instrument, typically used by businesses involved in global trade to ensure that payment of goods or services will be fulfilled between a buyer and a seller. L/Cs are standardized through the conventions established by the International Chamber of Commerce (in its UCP 600 guidelines).

In addition to the buyer and seller of the goods, an L/C also involves two intermediaries, typically banks, one acting for the buyer and one for the seller. The banks act as the guarantors in the transaction, ensuring that the seller is paid for its goods once they are shipped (and that the buyer is compensated if it does not receive the goods).

Historically, and even today, trade finance processes have been labor-intensive and document-centric, involving manual processing of large, complex contract documents. For our case study, a fictitious fintech company has seen the potential to improve efficiency and transparency by digitizing large parts of the process and is plans to create an online service, called Trade Finance eXchange (TFX), that will provide a digital platform for issuing and processing letters of credit. The company's strategy is to find a large international financial institution as its first customer and then sell access to the platform to all of that institution's trade finance clients.

In the longer term, the plan is to offer TFX as a white label (rebrandable) solution for large international banks in the trade finance business. Hence, in the short term, it will be a single-tenant system, but eventually, if this strategy is successful, it will need to grow to be a multitenant system.

The purpose of TFX is to allow organizations, engaged in writing and using L/Cs in the trade finance process, to interact in an efficient, transparent, reliable, and safe manner.

As outlined in the use cases in [Chapter 1](#), the key features of TFX are to allow international buyers of goods to request L/Cs from their banks and allow sellers of goods to redeem L/Cs for payment from their banks. For the banks, the platform acts as a simple and efficient mechanism for writing, executing, and managing L/C contracts with their clients.

The TFX system is being built as a cloud-native Software as a Service (SaaS) platform. For the purposes of our example, we use Amazon Web Services (AWS) as the cloud provider for the system, but though we reference AWS-specific features in the architectural description that follows, it is just an example of the type of cloud platform that a system like TFX is likely to be hosted on rather than an important aspect of the case study. We chose AWS because of its widespread adoption in the industry and assume that many technologists are familiar with its key services. We are not recommending AWS or its particular services in preference to other cloud providers, but we use AWS and its services to bring some realism to the case study.

## The Architectural Description

This appendix presents an outline architectural description for the first version of the TFX platform. In the spirit of normal industrial practice, we provide “just enough architecture” to explain the key aspects of the system without defining design details that are better made during implementation and described as code rather than a model. It is also important to remember that this is a model of a potential system and does not reflect a system we have built. Therefore, there are tradeoffs and decisions that would normally be necessary to meet delivery timelines or cater to particular product features that we have not been able to realistically include. For example, though do believe that tracking technical debt is an essential architectural activity, we can only provide a few hypothetical examples in this chapter.

The architectural description is also deliberately incomplete. Some aspects of it are left as open questions (outlined later in this appendix), which are then addressed in chapters, to illustrate some aspect of the architectural design process. Other aspects are simply left undefined, as they are not needed to support the content of the book, and so we omit them for space reasons.

The architectural description is represented as a set of views, taken from the Rozanski and Woods (R&W) set,<sup>1</sup> each illustrating a different aspect of the architecture. A full description of architectural viewpoints can be found on Wikipedia, and the R&W set in particular is explained in *Software Systems*

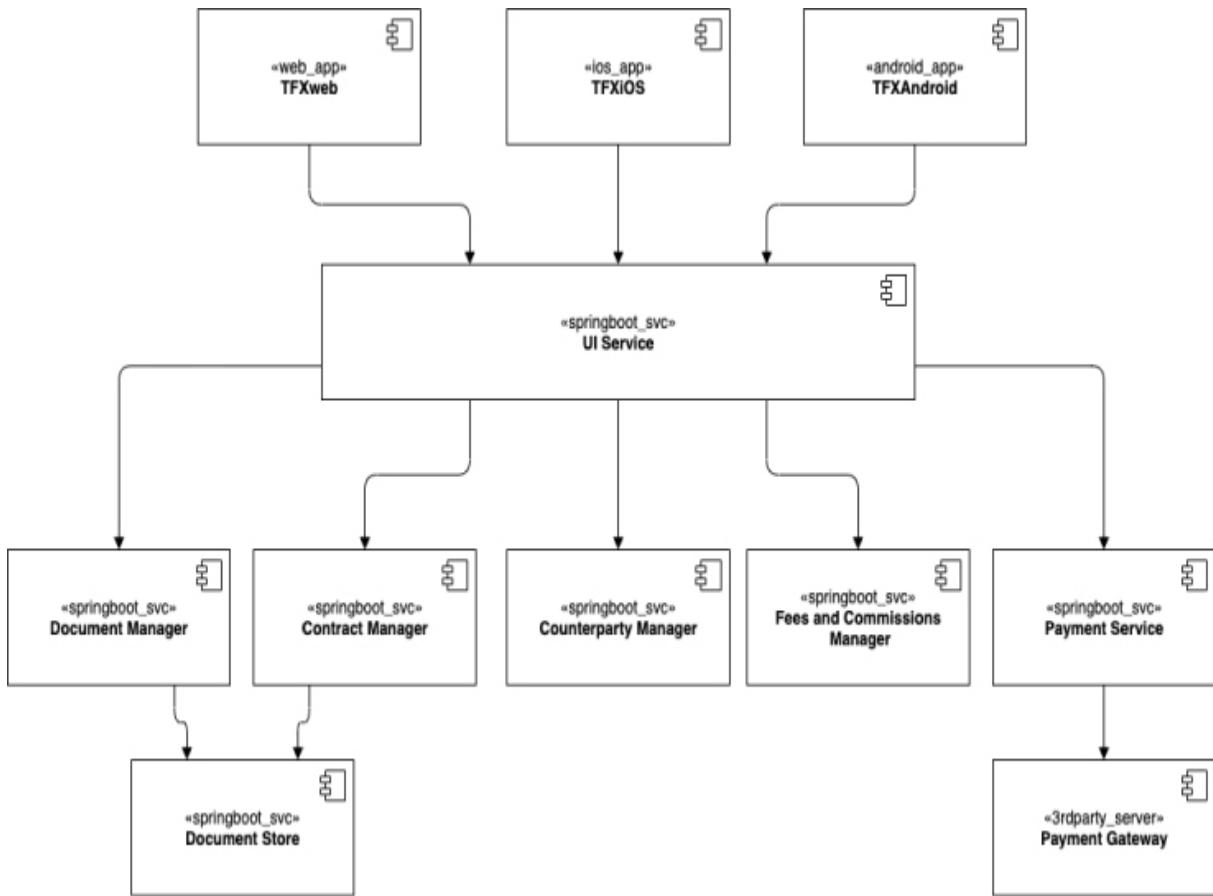
*Architecture* and at [www.viewpoints-and-perspectives.info](http://www.viewpoints-and-perspectives.info). Briefly, the views used in this model are as follows:

<sup>1</sup> Nick Rozanski and Eoin Woods, *Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives*, 2nd ed. (Addison-Wesley, 2012).

- *Functional*: The runtime functional structure of the system, showing the runtime functional components of the system, the interfaces that each exposes, and the dependencies and interactions among the elements. (Functional components are those that perform problem-domain-related activity rather than infrastructure-related activity, such as deployment, container orchestration, and monitoring).
- *Information*: The information structure of the system, showing the key information entities and their relationships and any significant state or life cycle models to illustrate how they are processed within the system.
- *Deployment*: Defines the runtime environment in which the system is intended to run, including the platform the system needs (compute, storage, networking), the technical environment requirements for each node (or node type) in the system, and the mapping of the software elements to the runtime environment that will execute them. As mentioned, we assume a cloud native application running on AWS.

## The Functional View

The functional structure of the system is illustrated by the Unified Modeling Language (UML) component diagram in [Figure A.1](#) and is explained in the following text.



**Figure A.1** Functional structure

In [Figure A.1](#), we use a simple boxes-and-lines notation broadly inspired by the UML2 component diagram.

- The rectangular shapes are functional components.
- The lines between components show which components invoke operations on which other components.
- The smaller text within the shapes, between guillemot brackets, are stereotypes. The stereotypes are used to indicate the type of each component. We use this mechanism to indicate how the component will be realized technically.

In [Table A.1](#), we explain the characteristics of each component informally in natural language. We also use the term *API* (application programming interface) throughout to indicate an intercomponent interface providing

callable operations, without necessarily defining what sort of API technology is to be used for each.

**Table A.1** Functional Component Descriptions

<b>Component: TFXWeb</b>	
Description	A conventional Web user interface designed to run in a desktop browser; writes using a collection of JavaScript, HTML, and Cascading Style Sheets (CSS)-based technologies
Responsibilities	<p>Provide a task-oriented interface to allow the requesting, creation, monitoring, execution, and management of letters of credit and other L/C artifacts</p> <p>Provide a role-specific interface for the different bank staff and client user types</p>
Provided Interfaces	Provides a human interface via a browser
Interactions	Calls the UI Service extensively to retrieve all of the information it needs and to make all of the updates required
<b>Components: TFXiOS and TFXAndroid</b>	
Description	Native mobile user interfaces optimized for use on mobile devices (both small and large screen)
Responsibilities	<p>Provide a task-oriented interface to allow the requesting, creation, monitoring, execution, and management of letters of credit and other L/C artifacts</p> <p>Provide a role-specific interface for the different bank staff and bank client user types</p>
Provided Interfaces	Provides a human interface via a mobile device
Interactions	Calls the UI Service extensively to retrieve all of the information it needs and to make all of the updates required

<b>Component UI Service</b>	
Description	An API providing a set of services to support the needs of the application's user interfaces  Acts as a façade for the domain services so is more functional than a simple API gateway
Responsibilities	Expose a set of domain services specialized to the needs of the user interfaces to allow retrieval of all of the domain entities from the system and their controlled update
Provided Interfaces	Provides an API to the domain services
Interactions	Calls the domain services to provide access to the domain entities and associated operations  Called extensively by the TFXWeb, TFXiOS, and TFXAndroid elements
<b>Component: Document Manager</b>	
Description	Service that provides a straightforward ability to perform operations on large, unstructured documents by other elements, particularly the user interfaces
Responsibilities	Accept files in a range of formats (text, JSON, PDF, JPEG)  Automatically generate metadata from files  Allow metadata to be attached to files  Allow the document set to be queried to return lists of matching documents, including natural language keyword indexes for nontextual formats such as PDF and JPEG  Allow specific document files to be retrieved
Provided Interfaces	Provides an API that allows two aspects of document management: (1) importing and retrieving large document files in various formats in single operations or multipiece operations; (2) management of the set of documents via metadata, allowing tagging, search, and retrieval of lists of documents matching complex criteria (e.g., dates and tags)
Interactions	Uses the Document Store to store, index, and manage the documents being imported

<b>Component: Counterparty Manager</b>	
Description	<p>Manages and provides controlled access to the information describing</p> <ul style="list-style-type: none"><li>• <i>Counterparties</i>—the banks and the buyers and sellers that the banks support</li></ul>

	<ul style="list-style-type: none"> <li>• <i>Bank accounts</i>—the bank accounts used by the banks and their clients</li> <li>• <i>Users</i>—information about the staff members of banks and their clients who are users of the system that cannot be held in the authorization and authentication systems</li> </ul>
Responsibilities	<p>Controlled creation, modification, and removal of counterparties</p> <p>Controlled creation, modification, and removal of bank accounts</p> <p>Controlled creation, modification, and removal of users</p> <p>Controlled creation, modification, and removal of user authorizations</p> <p>Persistent storage and maintenance of integrity of all of this information</p> <p>Simple retrieval of information related to counterparty, bank account, user, and authorization</p> <p>Complex queries on user and authorization information</p> <p>Checking of predicate-based authorization related queries (e.g., “is-authorized-to-release”)</p>
Provided Interfaces	Provides an API for use by internal system elements
Interactions	None
<b>Component: Contract Manager</b>	
Description	Provides the ability to create and manage contract objects that represent letters of credit (L/Cs)
Responsibilities	<p>Creation and management of contract domain objects</p> <p>Management of the state of a contract object (so that it is in a valid, clearly defined life cycle state at any time and only transitions between states according to a defined state model)</p> <p>Handling of required approvals for contract domain objects (which are multiple and can be complex)</p>
Provided Interfaces	Provides an API to allow Contract objects to be created, managed,

	approved, and queried
Interactions	<p>Depends on the Document Manager to store and retrieve document files (e.g., PDFs)</p> <p>Depends on the Fees and Commissions Manager to calculate fees and commissions as part of the L/C creation process</p> <p>Depends on the Counterparty Manager to record client account</p>

	transactions resulting from L/C redemption processes
<b>Component: Fees and Commissions Manager</b>	
Description	<p>Provides calculation operations to allow fees and commissions to be calculated on demand</p> <p>Allows rules for fee and commission calculations to be defined and managed</p>
Responsibilities	<p>Provide controlled creation, modification, and deletion of fee and commission rules</p> <p>Persistent storage and maintenance of integrity of the fee and commission rules</p> <p>Allow values of fees and commissions to be calculated on demand</p>
Provided Interfaces	<p>Provides an API to allow fee and commission rules to be defined</p> <p>Provides an API to allow fees and commissions to be calculated for specific transactions</p>
Interactions	None
<b>Component: Payment Service</b>	
Description	Provides the ability to create and manage payment domain objects
Responsibilities	Allows a payment to be created, approved if required, dispatched to a network, and its state queried
Provided Interfaces	Provides an API to allow payments to be created, managed, and queried
Interactions	Sends instructions to the Payment Gateway
<b>Component: Document Store</b>	
Description	Document store for unstructured document data
Responsibilities	<p>Provide controlled creation, modification, and deletion of unstructured document data with associated metadata</p> <p>Provide indexing of stored documents</p> <p>Provide tagging of stored documents</p> <p>Allow retrieval of documents by ID</p>

	<p>Allow document identification by metadata and tag search</p> <p>Allow document identification by full-text search</p>
Provided Interfaces	Provides an API to allow document storage, retrieval, queries, and management
Interactions	None
<b>Component: Payment Gateway</b>	

Description	A network service interface to allow standardized and simplified access to payment networks via a standard interface
Responsibilities	<p>Abstract the details of payment network access to a standard model</p> <p>Provide the ability for other system elements to perform standard payment network operations (pay, cancel, report)</p>
Provided Interfaces	Provides an API that provides simple and standardized access to payment network operations in a payment-network-independent manner
Interactions	None internally—accesses one or more external payment networks via their proprietary interfaces

## ***Usage Scenarios***

The general interactions between system elements expected in the architecture are as follows:

- The user interfaces call the UI Service for all interaction with the rest of the system. The service provides a set of APIs optimized for the needs of the user interfaces and so is acting as a façade to the rest of the system.
- When the user interfaces need to manipulate the key domain entities in the system, they call UI Service APIs, which in turn call the Counterparty Manager, Fees and Commissions Manager, Document Manager, Contract Manager, and Payment Service.
- More complex operations are implemented in the relevant domain service (e.g., the Contract Manager, which is responsible for creation

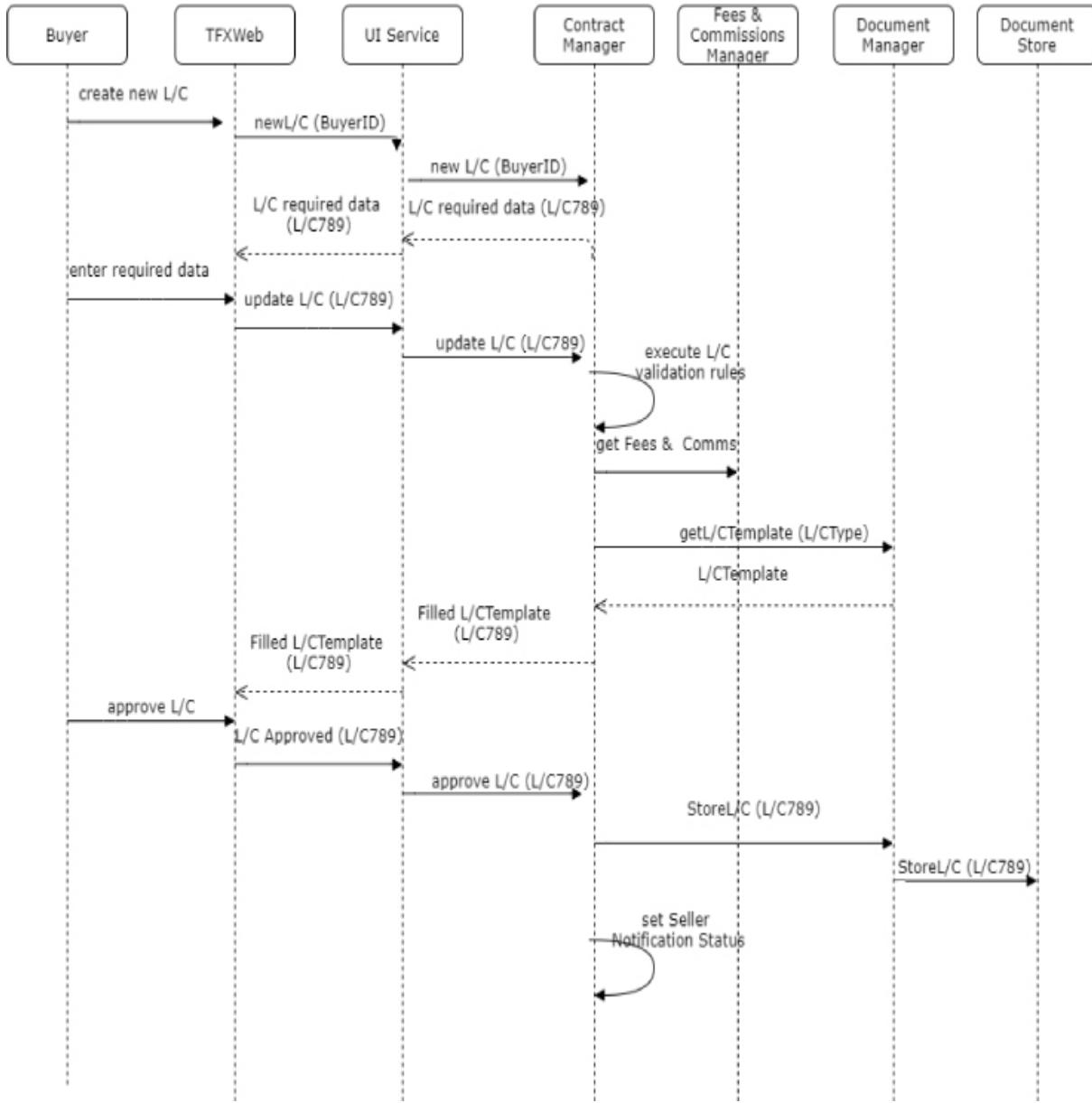
and export of an L/C contract, which is a complex process).

- Given the need for some long-running operations, some of the domain services will need to provide asynchronous as well as simple transactional API interfaces.
- When large documents need to be imported and processed (e.g., natural language processing), then a request is made from the UI Service to the Document Manager, which is responsible for this task.

These operations are inherently asynchronous, and when a request is made through the Document Importer API, a document ID is returned. The Document Importer writes the results of the process into the Document Manager, identified by that ID, and the caller can query the status of the document using the API and the document ID. The document ID is also used for multistage operations, such as uploading a document in pieces because of its size.

It is highly likely that the TFX system will also expose its APIs externally to business partners. This can potentially lead to different revenue streams and business plans. We have not expanded further on this aspect of TFX, but the most likely approach would be to create an external API service acting as a façade over the domain services, similar to the UI Service for our user interfaces.

To provide a tangible example of the system processing a request, consider the approval of the issuance of a L/C contract by an agent who is an employee of the issuing bank. The process is illustrated by the interaction diagram in [Figure A.2](#) and explained in the text.



**Figure A.2** Create L/C scenario—buyer

For the interaction diagrams in our case study, we have used a simplified version of UML notation. The boxes at the top of the diagram represent the runtime system elements that are communicating, and the vertical lines projecting down the page from them are their “lifelines,” showing where messages are received and sent by them, with the passage of time being represented by vertical distance. The solid arrows are invocations of some form between elements, their labels indicating the nature of the interaction.

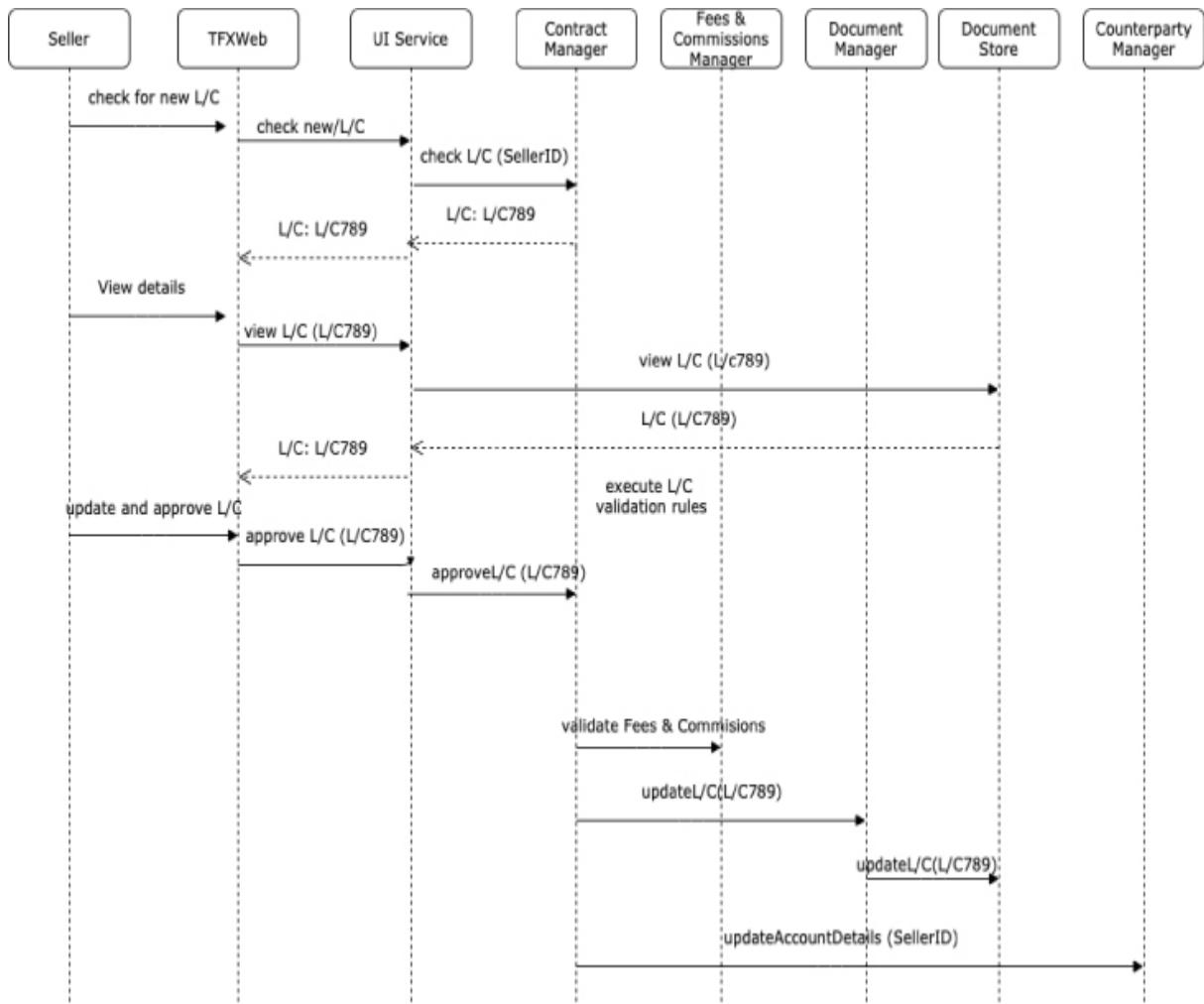
The dashed lines are responses from elements to a request or stimulus. We have used occasional stereotypes to indicate the type of an interaction.

The first scenario, illustrated by the interaction diagram in [Figure A.2](#), shows how a new L/C is created from the buyer's point of view.

The interactions in [Figure A.2](#) are as follows:

1. The buyer requests that a new L/C be created through TFXWeb.
2. The UI Service passes the request to the Contract Manager with the BuyerID.
3. The Contract Manager determines the data required for an L/C based on the BuyerID—this assumes the system is configurable based on the characteristics of the buyer. The Contract Manager also assigns a unique ID (789) to the request.
4. The requested data is presented in the user interface, and the buyer fills in all the data (e.g., seller details, contractual terms)
5. The buyer enters all the required data, which is then passed to the Contract Manager.
6. The Contract Manager executes validation rules, including fees and commissions via the Fees and Commissions Manager.
7. The Contract Manager requests the L/C template from the Document Manager.
8. The L/C template filled in with all the relevant data is presented to the buyer for final validation.
9. The buyer approves the L/C, which is passed on to the Contract Manager.
10. The Contract Manager asks the Document Manager to store the L/C. The Document Manager extracts all relevant metadata and stores the L/C in the Document Store.
11. The Contract Manager sets the notification status of the seller to indicate that an L/C is set up for the seller.

The same scenario illustrated from the seller's perspective is shown in [Figure A.3](#).

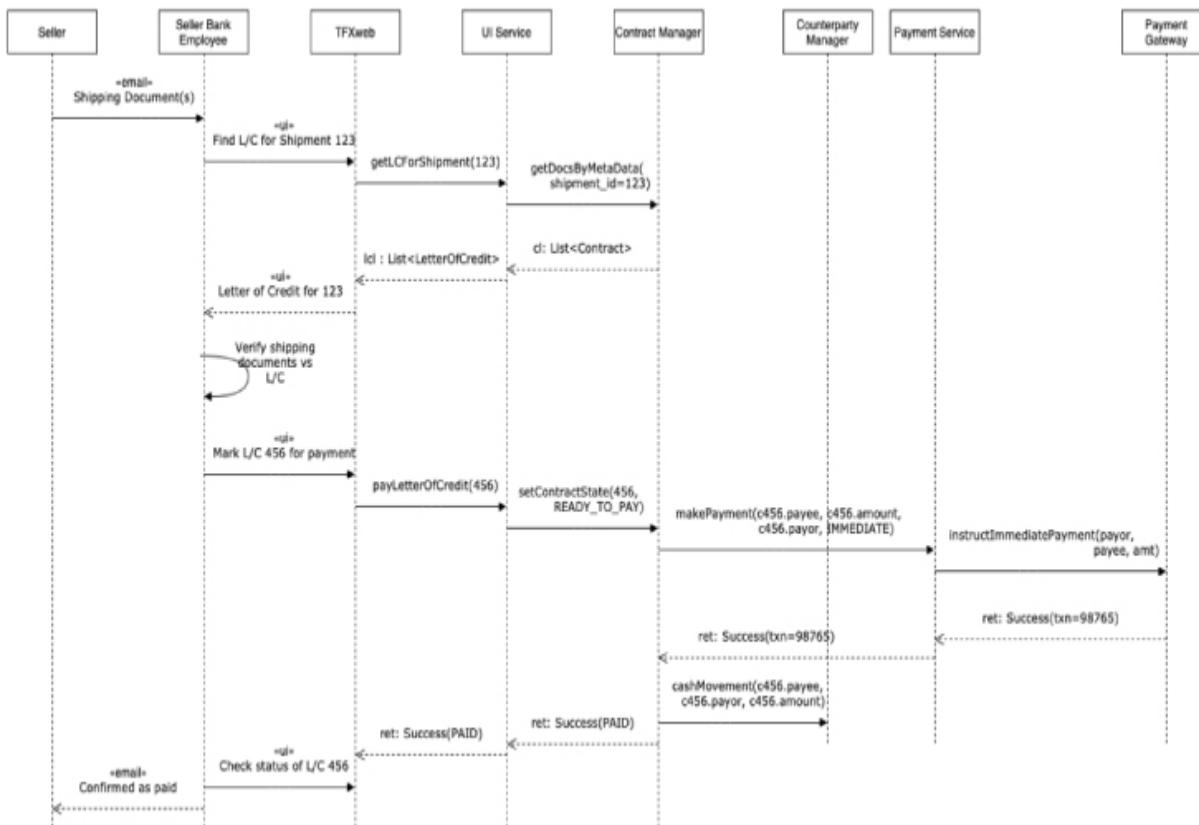


**Figure A.3** Create L/C scenario—seller

1. The seller requests checks for new L/C via TFXWeb.
2. The UI Service passes the request to the Contract Manager with the SellerID.
3. The Contract Manager locates the new L/C set up by the buyer, and this information is presented to the Seller.
4. The Seller requests to see the full L/C document.
5. The UI Service passes the request to the Document Store with the L/C ID.
6. The Seller approves the L/C, which the UI Service passes to the Contract Manager.

7. The Contract Manager executes validation rules, including validating fees and commissions via the Fees and Commissions Manager. For simplicity, we do not show any exceptions or feedback loops that would be present in a more complex scenario; we present only the simplest “happy day” scenario here.
8. The Contract Manager asks the Document Manager to store the updated contract with seller approval information in the Document Store. The Document Manager updates relevant metadata for the Contract.
9. The Contract Manager asks the Counterparty Manager to update the account details of the seller.

A second example is the arrival of goods, triggering the payment of the supplier according to the terms of the L/C contract. This process is illustrated in [Figure A.4](#) and explained in the text.



**Figure A.4** Paying a letter of credit

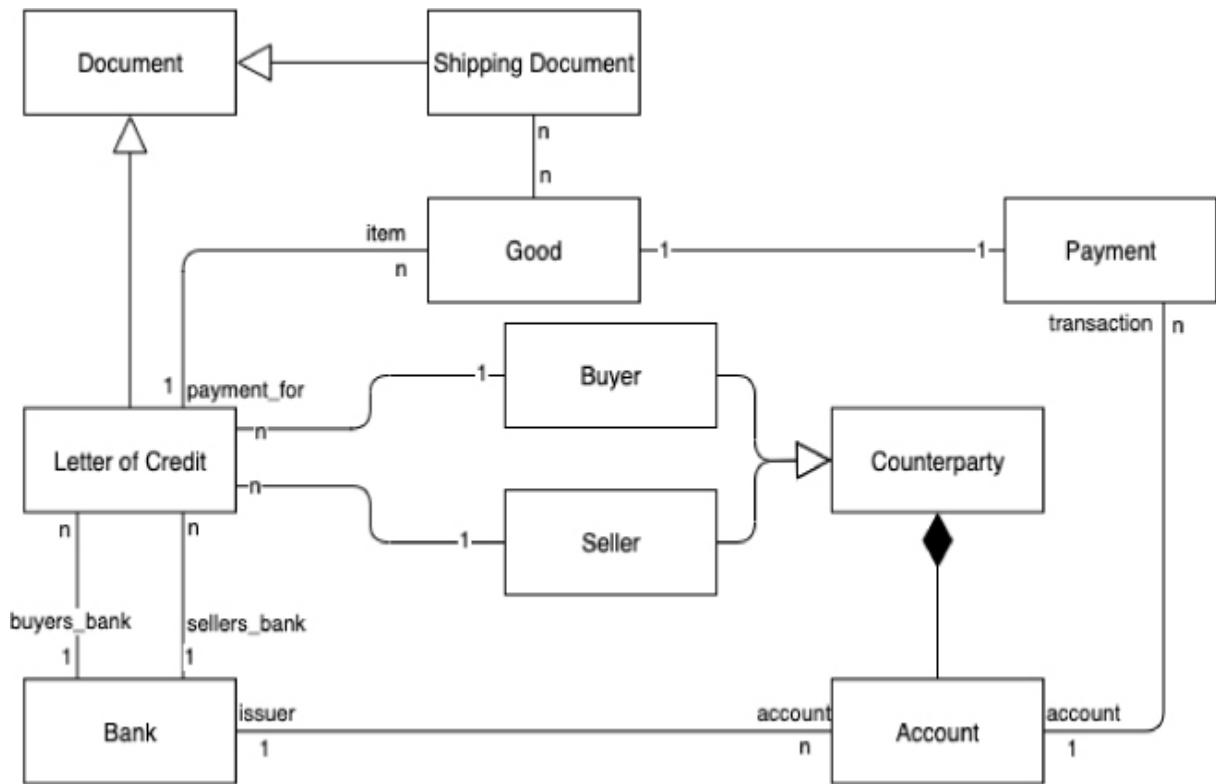
1. A bank employee is notified by a seller of the shipment of a consignment of goods that are guaranteed under an L/C, probably through a bill of lading document via email.
2. The employee logs onto the TFXWeb user interface and searches for the relevant L/C (probably using a consignment number).
3. The TFXWeb interface calls the UI Service to provide a straightforward interface to this operation.
4. The UI Service calls the Contract Manager, passing it the search criteria, and the Contract Manager returns details of any contracts that match the criteria.
5. The TFXWeb interface displays the contracts returned and allows the user to inspect their content. The user finds the relevant contract and checks the conditions that the L/C requires to release payment.
6. The user confirms satisfaction that the required conditions are met (by checking the documents that have been provided and perhaps verifying some details by telephone or email). Parts of this process have the potential for automation, which we don't show here.
7. Once satisfied, the user selects the contract in the TFXWeb user interface and marks it as ready for payment.
8. The TFXWeb user interface calls the UI Service, which calls the Contract Manager to indicate this change of status via an API call.
9. The Contract Manager retrieves the details of the contract and creates a payment request for the currency and amount from the bank's account to the recipient details, all as specified in the contract. It then calls the Payment Service to instruct that payment be made and sets the contract status to "payment requested."
10. The Payment Service receives and saves the payment request. It then creates a payment instruction for the Payment Gateway and sends it to the Payment Gateway.
11. The Payment Gateway attempts to make the payment and sends a success or failure status back to the Payment Service.

12. If the payment is successful, the Payment Service marks the payment as completed and calls the Contract Service to indicate this event.
13. The Contract Manager updates the payor and payee accounts via the Counterparty Manager to indicate the cash movement for the transaction.
14. The Contract Service marks the contract as paid and completed, and success is returned to the UI Service, then in turn to the TFXWeb service, where the bank employee can see the status and notify the seller.

Should the payment instruction fail with a transitory error, the Payment Service would retry a small number of times. If the payment instruction fails with a permanent error or multiple transitory errors, the Payment Service will mark the request as failed and call the Contract Manager to indicate this event. The Contract Manager would then mark the L/C contract as in an error state and needing operational attention to rectify the payment information.

## Information View

In the Information view, we describe the key data structures for the system and how they relate to each other. The data model for the system is shown in [Figure A.5](#).



**Figure A.5** Information view—data model

There are a number of different conventions for representing data models, but we based ours on UML2. The rectangular shapes are entities (or “classes”); the lines indicate relationships, and the role of each entity is specified at the entity’s end of the relationship, where the role is not obvious, along with the cardinality of that end of the relationship (1 meaning exactly 1 and n meaning between 0 and n). The relationship with an open triangle at one end indicates a generalization relationship (“inheritance”), and a filled diamond indicates an aggregation (“containment”) relationship.

The key parts of the information structure in the system are as follows:

- *Letter of Credit*: The L/C is the central information entity in the system and its reason for existing. It is a type of contract to remove the risk that the buyer of some goods (the applicant) does not pay the seller of the goods (the beneficiary).
- *Shipping Document*: A document used to prove that goods have been shipped, and therefore a L/C for those goods may be redeemed. It

normally takes the form of a bill of lading (or bill of material) document.

- *Document*: A generalization of the L/C and shipping document, representing the generic aspects of a document that are not specific to the characteristics of either.
- *Payment*: The payment made to the seller on proven dispatch of the goods.
- *Good*: A representation of the good that is being traded between seller and buyer and that is being guaranteed via the L/C.
- *Buyer and Seller*: Represent the participants in the contract, both of which are types of counterparty that have a client relationship with a bank.
  - The *Seller* is the counterparty whose payment is being guaranteed by the contract. This is usually an exporter in international trade.
  - The *Buyer* is the counterparty who requests the contract in order to have the seller release the goods before payment. This is usually an importer in international trade.
  - The *Buyer's Bank* is the financial institution that creates the contract and represents the buyer.
  - The *Seller's Bank* is the financial institution that represents the seller and advises the seller when credit has been received.
- *Bank*: Represents the banks involved in the L/C process that create and redeem the L/C and provide their clients with account holding and payment services.
- *Account*: The bank accounts associated with the counterparties used to facilitate the transaction, which may be held in different institutions, countries, and currencies.

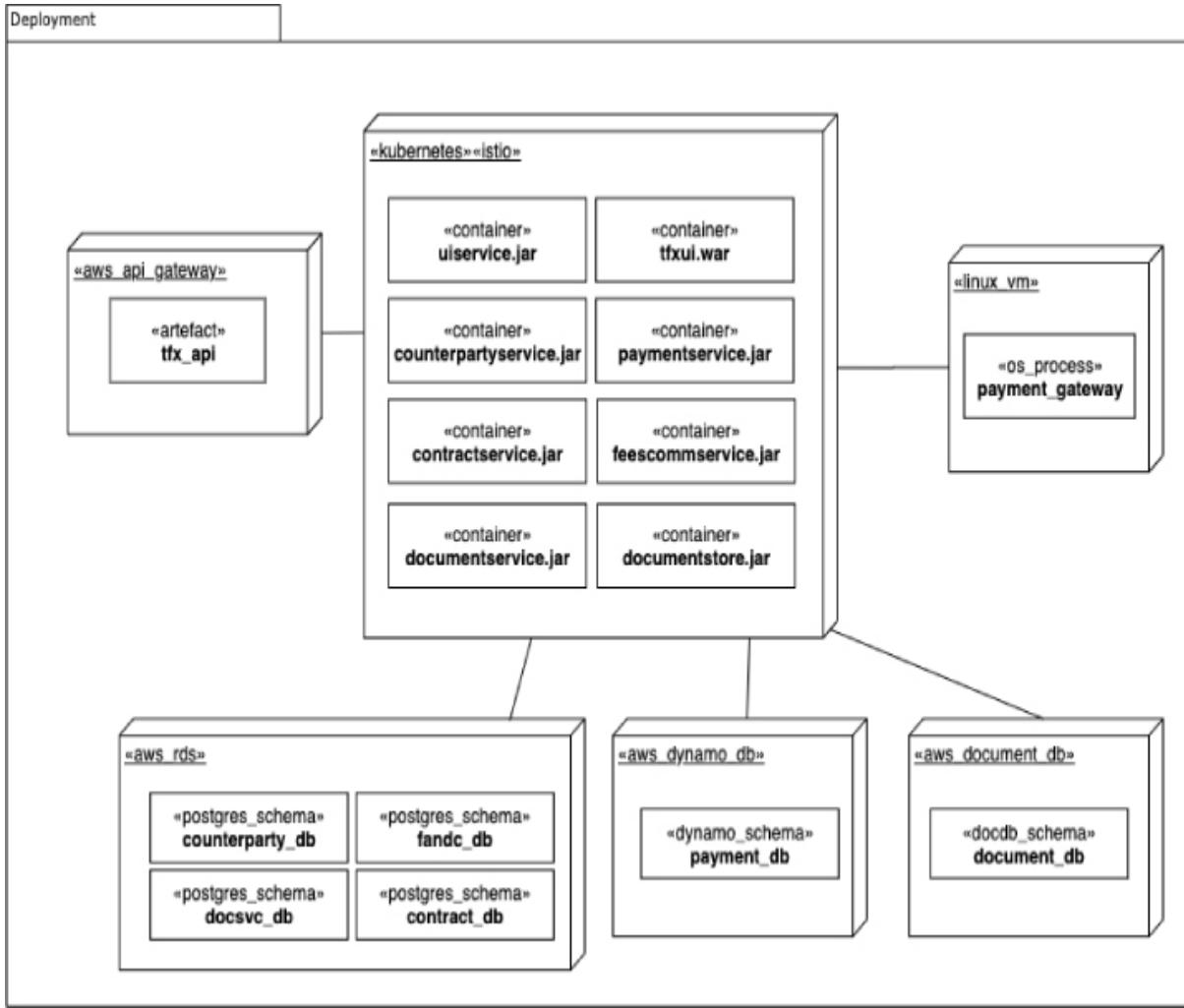
We are aware that this is a rather simplified view of the information structure needed to support a realistic L/C transaction, but we feel that it captures its essence and serves our purposes while avoiding excessive complexity.

We have repeatedly found that, in practice, models like this are most valuable when they capture the essence of the domain rather than try to capture all of the detail that a domain expert would know. This sort of model helps to explain the essentials of the domain to people who are new to it and provides a starting point for learning and a context into which more detailed knowledge can be placed as it is learned.

A model like this can also provide a starting point for creating the more detailed domain model and ubiquitous language needed by a domain-driven design approach. As soon as implementation begins, the model will need to be refined (e.g., in this case, “buyer” and “seller” might become roles for the “counterparty” rather than entities in their own right, and missing elements, such as fee and commission data, will need to be added). However, we have found that collaborating with the domain experts to produce a fairly abstract model like this one is a great first step in the process, and such a model continues to be a useful reference point for the project as it evolves.

## **Deployment View**

The deployment of the system to its operational environment is described in the UML deployment diagram in [Figure A.6](#).



**Figure A.6 Deployment view**

We are assuming that TFX will be deployed on the AWS cloud platform, although we have taken care not to rely on any very specific service that AWS provides, leaving the possibility of using other cloud providers or on-premises deployment later if needed.

The application requires the following services from its deployment environment:

- A *container execution environment* such as Kubernetes and some sort of service-level infrastructure provider for service discovery, request routing, security, and request tracing such as the Istio service mesh. This is shown in the diagram as the execution environment components stereotyped as «kubernetes» and «istio».

- A *database service* providing SQL relational database storage. This is shown in the diagram by the execution environment stereotyped as «aws\_rds» (the AWS Relational Database Service).
- A *database service* providing document-oriented database storage. This is shown in the diagram by the execution environment stereotyped as «aws\_document\_db» (the AWS DocumentDB service)
- A database service providing key–value oriented database storage. This is shown in the diagram by the execution environment stereotyped as «aws\_dynamo\_db» (the AWS DynamoDB service).
- *Linux VMs*, shown in the diagram as the execution environment stereotyped as «linux\_vm» (corresponding to the AWS EC2 virtual machine service).
- An *API gateway* to provide security and load management for our application services when accessed from the Internet. This is shown in the diagram as the execution environment stereotyped as «aws\_api\_gateway» (the AWS API Gateway service).

Most of the system is deployed as a set of containerized services running in a Kubernetes deployment environment using Istio to act as a service mesh and provide standardized service-level infrastructure services, in particular routing, discovery, and security.

The databases for each of the services will use the managed database services of AWS, specifically the Postgres flavor of Amazon RDS, the DocumentDB service, and the DynamoDB database service.

Linux VMs are required to run the Payment Gateway component because this is third-party binary software that is not normally run in a container, and its installation and management facilities assume that it is running as a regular operating system process on a Linux machine.

The AWS API Gateway will be used to provide the external API endpoints to support the mobile applications (and any future public application API required). The WebUI will use the same service APIs as the mobile applications but will access the services directly rather than via the API Gateway.

The only component that needs to initiate outbound network requests is the Payment Gateway, which needs to contact the payment network gateway using HTTPS. The other elements do not need to send requests outside the application. Hence, networking can be restricted to inbound requests via the API Gateway and outbound requests from the Payment Gateway.

Clearly, the cloud platform and TFX’s requirements of it will evolve over time, so the choices outlined here will change as TFX and the cloud platform develop. However, the foundational choices of how the system is deployed to the cloud platform are unlikely to vary dramatically for the foreseeable future.

## Architectural Decisions

In [Chapter 2](#), “[Architecture in Practice: Essential Activities](#),” we state that architectural decisions are the unit of work of architecture, so it would be remiss of us not to describe the architectural decisions for our case study. However, in the space we have available, it is not possible to list all of the architectural decisions for building a system such as TFX. Consequently, we present a selection of architectural decisions in [Table A.2](#). The foundational decisions (FDNs) support the case study as described in this appendix. The additional decisions provide examples of the evolution of the system based on topics covered in [Chapters 3 through 8](#).

**Table A.2** Architectural Decision Log

Type	Name	ID	Brief Description	Options	Rationale
Foundational	Native Mobile Apps	FDN-1	The user interface on mobile devices will be implemented as native iOS and Android applications.	Option 1, Develop native applications  Option 2, Implement a responsive design via a browser	Better end-user experience. Better platform integration. However, there is duplicated effort for the two platforms and possible inconsistency across platforms.
Foundational	Domain-Oriented Structure	FDN-2	The system's functional structure is based on the structure of the domain (contracts, fees, documents, payments) rather than structured around processes or transactions (e.g., L/C creation, L/C closure, payments release).	Option 1, Domain-oriented structure  Option 2, Process/transaction-oriented structure	The stored state of the system is partitioned into separate cohesive stores to minimize coupling.  The services of the system are cohesive with well-defined responsibilities. The services of the system can be combined in different ways for different processes or transactions.  However, more interservice communication is needed for many processes than if the system were structured around processes.
Foundational	Service-Based Structure	FDN-3	The system's fundamental structuring unit is a service, accessed via an API (synchronously) or a message request (asynchronously). It is not structured around events or any other mechanism.	Option 1, Conventional monolithic structure  Option 2, Event-driven architecture (EDA)  Option 3, Service-based architecture	Services are well understood by most people and fairly straightforward to design.  Monitoring and debugging are much easier than with implicit connections such as event streams and with event-driven systems. A set of services is easier than a monolith to evolve quickly and release rapidly. The tradeoff is that, over time, many interservice dependencies will emerge,

				making some types of evolution more difficult (e.g., replacing a service)
Foundational	Independent Services	FDN-4	<p>The system is structured as a set of independent runtime services, communicating only through well-defined interfaces (rather than shared databases or as monolithic collections via local procedure calls). In this way it is structured in the spirit, if not all the detail, of the microservices pattern.</p>	<p>Option 1, Independent services</p> <p>Option 2, Conventional monolithic structure</p> <p>Option 3, Services with shared databases</p> <p>Option 4, Event-driven architecture</p> <p>Evolution and deployment should be easier because of localized change, and we will avoid interservice coupling in the data layer. Different teams will be able to make different implementation decisions (e.g., database technology) for different services if this is deemed to be sensible. The system can evolve more quickly than a monolith system. The tradeoff is that there will be more deployable units to deploy, monitor, and operate, and operational monitoring and debugging are</p>

				likely to be more difficult than with more monolithic structures.
Foundational	RPCs for Service Interaction	FDN-5	In the current implementation, all of the interelement communication will use remote procedure calls (RPCs) of some sort (most of them being REST-style JSON/HTTP invocations) rather than using command and query messages to implement request/response interactions over messaging.	Option 1, RPC-based communication Option 2, Message-based communication
Foundational	Generic Cloud Services	FDN-6	In deciding how to deploy the application, the team chose to use AWS but to deliberately use cloud services that are easily procured on other clouds too (e.g., Azure).	Option 1, AWS-specific architecture using AWS specific services Option 2, Intentionally designing to avoid AWS specific services
Foundational	Containerization, Kubernetes, and Istio	FDN-7	Given the mainstream adoption of Docker-based containers and the Kubernetes container orchestration system, the team chose to package the services and Web UI as containers, running in	Option 1, simple operating system level deployment of binaries Option 2, Use Kubernetes and containers

		Kubernetes. They also assume a service mesh middleware (e.g., Istio) is used for service discovery, interservice security, request routing, and monitoring.	Option 3, Use containers and an alternative such as DC/OS or Nomad instead of running K8S ourselves, which simplifies it considerably. It will still involve installing Helm and Istio manually and configuring and using the resulting environment.
Data	RDMS for Transactional Services	DAT-1	<p>Databases for the Counterparty Manager, Contract Manager, and Fees and Commissions Manager will be relational databases.</p> <p>Option 1, Relational database</p> <p>Meets quality attribute requirements.</p> <p>Option 2, Document store</p> <p>ACID transaction model aligns with access patterns.</p> <p>Option 3, Other</p> <p>Skill base is available in the</p>

				NoSQL choices	team.
Data	Database Technology for Document Store	DAT-2	The Document Store will utilize a document store for its persistence layer.	Option 1, Relational database Option 2, Document store Option 3, Other NoSQL choices	Supports natural structure of documents and extensible if required. Query structure (entire document) supports expected patterns.
Data	Database Technology for Payment Service	DAT-3	The Payment Service will utilize a key-value store for its persistence layer.	Option 1, Document store Option 2, Key-value store Option 3, Relational database	The actual Payment Service is external to the platform. Basic database that will act as a log and audit trail. Can meet future scalability requirements.
Data	Good Tracking Database	DAT-4	The Good Tracking Database will utilize a document store for its persistence layer	Option 1, Document store Option 2, Key-value store Option 3, Relational database	Will be able to meet evolving data model requirements. Supports developer ease of use. Sufficient to meet known scalability and availability requirements.
Security	Encrypted Traffic	SEC-1	Avoid privacy threats for internal system intercomponent requests by encrypting intercomponent communication.	Option 1, Rely on boundary security of the environment Option 2, Encrypt all intercomponent traffic with Transport Layer Security (TLS) Option 3, Encrypt sensitive information in request parameters	The TLS point-to-point solution (Option 2) is relatively easy to implement. Relying on attackers not reaching the network is unwise, particularly in private cloud. Encrypting sensitive parameters is more efficient than TLS but leaves a lot in the clear and is more complex to implement securely.
Security	Cloud Secret Store	SEC-2	Protect secrets, such as database login credentials, using cloud secret store	Option 1, Obfuscate in properties files Option 2, Use cloud	Option 1 is simple, nonintrusive, and quick to implement but not very secure

			secret store service	if an attacker gains access to the environment. Option 2 is more intrusive and more work but is necessary given the sensitivity of some of the data.
Scalability	Analytics DB	SCA-1	<p>As the TFX transactional database is updated, the updates are replicated to the analytics database, using the TFX DBMS replication mechanism process</p> <p>Option 1, Use the TFX DBMS Replication process</p> <p>Option 2, Use an event bus to replicate the updates</p>	<p>Using an event bus may increase latencies by milliseconds up to seconds at high volumes because of the data serialization, data transmission, data deserialization, and write processing. Using the TFX DBMS replication mechanism would reduce the propagation latency, as in most cases, it would be shipping database logs rather than processing events. This is very important for consistency and</p>

				scalability. The tradeoff is that it does increase coupling between the components.
Scalability	TFX Table Partitioning Implementation	SCA-2	Do not use table partitioning for the initial implementation of TFX	<p>Option 1, Implement table partitioning as part of the initial TFX installation</p> <p>Option 2, Defer implementation until multitenancy approaches other than the initial full replication are considered</p> <p>Because the initial TFX deployment will be a full replication installation, the team decides not to implement table partitioning at this time. This decision applies principle 3, "Delay design decisions until they are absolutely necessary." However, the team will need to reassess using table partitioning or even sharding when it becomes necessary to switch to other multitenancy approaches as the number of banks using TFX increases.</p>
Performance	Prioritize Requests	PRF-1	Use a synchronous communication model for high-priority requests such as L/C payment transactions. Use an asynchronous model with a queuing system for lower-priority requests such as high-volume queries.	<p>Option 1, Use a synchronous model for all requests</p> <p>Option 2, Use a synchronous communication model for high-priority requests and an asynchronous model for lower-priority ones</p> <p>Option 3, Use asynchronous model for all requests</p> <p>Using a synchronous model for all requests is easier to implement but is likely to create performance issues. Introducing an asynchronous model for lower-priority requests addresses those issues. This decision is an evolution to FDN-5.</p>
Performance	Reduce Overhead	PRF-2	Consolidate Counterparty Manager and Account Manager into a single	Originally, the Counterparty service was split into two components: a Counterparty

			service.	modifiability	Manager, which deals with the main data about a counterparty, and an Account Manager, which deals with the accounts of the counterparty. This could have easily been a viable alternative. However, the level of cross-component communication created by this approach would have been significant and likely to create performance bottlenecks. Early in the architectural design process, the team opted to consolidate both components into a Counterparty service to address performance concerns.
Performance	Caching	PRF-3	Consider using database	Option 1, Do not use	Using database object cache

			<p>object cache for the Counterparty Manager, the Contract Manager, and the Fees and Commissions Manager components.</p>	<p>database object cache to simplify the implementation of TFX</p> <p>Option 2, Use database object cache for specific TFX components</p>	<p>for specific TFX components addresses potential performance issues, but it complicates the implementation of those components, as using caching introduces design and programming challenges.</p>
Performance	Non-SQL Technology	PRF-4	<p>Use a document database for the Document Manager and a key-value database for the Payment Service.</p>	<p>Option 1, Use non-SQL technology for specific TFX databases</p> <p>Option 2, Use SQL technology for TFX databases</p>	<p>SQL database technology is easier to use and more familiar to the TFX developers. However, using it for TFX databases such as the Document Manager and Payment Service databases is likely to create performance issues. This supports DAT-2 and DAT-3.</p>
Resilience	Load Shedding	RES-1	<p>TFX will use the cloud API gateway to monitor total request volume and shed load when it exceeds a predefined threshold.</p>	<p>Option 1, Use load shedding via cloud API gateway</p> <p>Option 2, Don't use load shedding</p> <p>Option 3, Build load shedding into application</p>	<p>Load shedding will protect the application from overload without application changes, so it is valuable. Building it into the application is more flexible but requires a significant amount of engineering effort. API gateway judged to be good enough for current requirements. Also acts as a bulkhead in the application deployment.</p>
Resilience	UI Client Circuit Breakers	RES-2	<p>TFX user interfaces will all incorporate the Circuit Breaker pattern to ensure that they back off if errors are encountered.</p>	<p>Option 1, Use circuit breaker in clients</p> <p>Option 2, Use simple pause when error is encountered by client</p>	<p>A simple pause on error will overreact to a single transient service error but not provide the service with significant protection in case of a real failure.</p>

				failure. Ignoring errors risks complicating service recovery by continuing to send significant request traffic during restart.
Resilience	Health Check Functions	RES-3	All TFX runtime components will provide a standardized health check operation to return confirmation (or not) that the component is working correctly.	<p>Option 1, Use a standard health check operation</p> <p>Option 2, Use an external mechanism to check component operation</p> <p>Option 3, Rely on log files</p>
ML	TFX Document Classification Model	ML-1	TFX will leverage prepackaged reusable software for document classification.	Given the architecture tradeoffs involved in Option 1, such as maintainability, scalability, security, and ease of deployment, leveraging prepackaged reusable software is a better option. Option 1 would require significant build effort.

We have not represented the full architecture of the TFX system based on all the architectural decisions. This is on purpose; our objective is to provide an overview of tradeoffs that teams need to make.

Finally, we would expect that architectural decisions would have more detail and supporting material in a real-life system. Refer to [chapter 4](#) of *Continuous Architecture*<sup>2</sup> for a more comprehensive overview of this.

<sup>2</sup> Murat Erder and Pierre Pureur, *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World* (Morgan Kaufmann, 2015).

# Other Architectural Concerns

## Multitenancy

The plan for TFX involves an initial version that is used by the first client of the fintech creating it, a major financial institution involved in trade finance. This first client plans to use TFX to automate trade finance processes involved in issuing, managing, and processing L/Cs for their own customers. However, in the future, the fintech would like to offer the platform for trade finance processes to other financial institutions that offer a similar service.

The fintech does not want to enter the packaged software business, and therefore needs to offer the software as a SaaS deployment for other banks. This implies some form of multitenancy, as TFX will need to simultaneously host the workloads of a number of banks whose workloads must be totally isolated from each other.

There are at least three ways that the team could achieve this:

- *Full replication:* The entire system is replicated for each customer, each in its own isolated cloud environment. This is the simplest form of multitenancy, needing little or no changes to core application code, and for a small number of customers, it works well, eliminating most security risks, ensuring resource isolation between the clients, and allowing independent upgrade and support for each customer. The problem with this approach is when a large number of customers need to be supported, meaning a large number (tens or hundreds) of installations, all of which need to be monitored and managed independently. They also need to be upgraded reasonably quickly to avoid huge operational complexity of having many versions in production simultaneously (and sometimes quickly for situations such as security patches).
- *Database replication:* Involves having a single set of application services but a separate set of databases for each customer. Hence, each customer's data is physically isolated, but all of the incoming requests are processed by a single set of application services. This approach offers many customers considerable reassurance that their data will not

leak to other customers and, for large numbers of customers, is operationally simpler than full replication. However, it requires fairly extensive changes to parts of the core application, as application services need to carry the current customer as part of the request context, and this needs to be used to select the right database for each operation. This approach reduces, but does not eliminate, operational overhead as the number of customers increases. A single set of application services significantly reduces the complexity of monitoring, upgrading, and patching, but upgrades may still involve work on a large number of databases when a large number of customers have adopted the system.

- *Full sharing:* The most complex approach technically but the most efficient operationally and for the long term. This is the approach used by the hyper-scalers, such as Google, Microsoft, and Amazon. A single system is used to support all of the customers, with all system elements being shared among them. This implies intermingled client data in databases as well as shared application services. The huge advantage of this approach (from the operator's perspective) is a single system to monitor, operate, patch, and upgrade. The disadvantage is that it requires sophisticated mechanisms within the application to ensure that a particular request is always processed in the correct security context and so can never access data from the wrong customer's security realm.

In summary, our initial deployment will be a full replication installation, as we have no customers and need to get to market as quickly as possible for our owning bank, but we will probably need to implement one of the more sophisticated approaches once the system needs to support a significant number of customers.

## Meeting Our Quality Attribute Requirements

If you are reading this appendix before the rest of the book, you may be wondering about all the other architectural concerns and mechanisms that we haven't mentioned in this chapter, such as logging and monitoring, security mechanisms (e.g., authentication), how we ensure resilience of the databases, and so on. We haven't forgotten about these concerns, but we discuss them in the context of how they contribute to the system's meeting

its quality attribute requirements, such as scalability, performance, security, and resilience, so you will find them discussed throughout the relevant chapters for those quality attributes.

# Appendix B. Comparison of Technical Implementations of Shared Ledgers

As we saw in [Chapter 8, “Software Architecture and Emerging Technologies,”](#) shared ledgers can be implemented as public blockchains, such as Bitcoin or Ethereum; private blockchains, such as Quorum; public distributed ledgers, such as Corda Network or Alastria; and private distributed ledgers, such as Corda R3. Private blockchains and distributed ledger technologies are usually operated by a consortium of parties with similar goals, such as banks, that want to improve their trade financing processes and decrease their risks.<sup>1</sup> [Table B.1](#) compares the capabilities of three common implementations from an architecture perspective.

<sup>1</sup> See “When Nontechnical Stakeholders Want to Introduce New Technologies” sidebar in [Chapter 8](#).

**Table B.1** Comparison of Three Blockchain and Distributed Ledger Technology (DLT) Implementations

<b>Element</b>	<b>Public/Permissionless Blockchain</b>	<b>Private/Permissioned Blockchain</b>	<b>Private Distributed Ledger</b>
Data	Redundant data persistence, as all data is stored across all nodes	Redundant data persistence, as all data is stored across all nodes	Shares data selectively across some nodes
Network access	Public	Private only	Private only
Network structure	Large autonomous and decentralized public network for anonymous peer-to-peer transactions	Collection of centrally planned private networks for identifiable peer-to-peer transactions	Collection of centrally planned private networks for identifiable peer-to-peer transactions
Network governance	Decentralized autonomy	Governed by a centralized organization or a consortium	Governed by a centralized organization or a consortium
Network trust	Trust is distributed among all network participants in aggregate.	Trust is distributed among all network participants. All transactions are validated before they are accepted. Network operator control is strictly limited.	Trust is distributed among all network participants. All transactions are validated before they are accepted. Network operator control is strictly limited.
Transaction validation	Transaction validation rules are defined by the protocol, and all nodes validate every transaction they receive against these rules (including when receiving a block from a miner).	Nodes validate transactions before accepting them.	Nodes validate transactions before accepting them.
Transaction confirmation	The decision to confirm a transaction is made by consensus. That is, the majority of computers in the network must agree that the transaction is confirmed (probabilistic confirmation).	The decision to confirm a transaction is made by a set of authorized parties.	A DLT confirms transactions without involving the entire chain, often by either using a notary node (e.g., Corda) or endorsers (e.g., Hyperledger Fabric). In Corda, notaries decide which transactions get confirmed.
Cryptocurrency	Natively available	Not available	Not available

security	Susceptible to 51% attacks <sup>2</sup> as well as to traditional attacks (e.g., compromising a private key would allow impersonation)	Susceptible to 51% attacks as well as to traditional attacks	Nodes (especially notary nodes) potentially susceptible to traditional attacks

<sup>2</sup> A *51% attack* is an attack on a blockchain by a group of miners controlling more than 50 percent of the network's computing power.

# Glossary

**2FA:** See [two-factor authentication](#).

**ACID transactions:** Transactions in a resource manager, typically a database, that guarantee an atomic transition from one state to another that is atomic, consistent, isolated, and durable, hence “ACID.”

**architectural runway:** In the SAFe methodology, with architectural features implemented in a program, the enterprise incrementally builds an architectural runway. Instead of big up-front architecture, agile teams develop the technology platforms to support business needs during each release. “The Architectural Runway consists of the existing code, components, and technical infrastructure needed to implement near-term features without excessive redesign and delay.”<sup>1</sup>

<sup>1</sup> SAFe, “Architectural Runway.” <https://www.scaledagileframework.com/architectural-runway>

**architecturally significant:** An architecturally significant requirement (scenario) is a requirement (scenario) that will have a profound effect on the architecture—that is, the architecture might well be dramatically different in the absence of such a requirement (scenario).<sup>2</sup>

<sup>2</sup> Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice*, 3rd ed. (Addison-Wesley, 2012), 291.

**ATAM:** The architecture tradeoff analysis method (ATAM) is a risk-mitigation process used early in the software development life cycle. ATAM was developed by the Software Engineering Institute at Carnegie Mellon University. Its purpose is to help choose a suitable architecture for a software system by discovering tradeoffs and sensitivity points.

**ATAM utility tree:** In ATAM, utility trees are used to document and organize quality attribute requirements. They are used to prioritize the requirements and to evaluate the suitability of an architecture against its quality attribute requirements.

**attack vector:** A means by which an adversary can, given the right skills, conditions, and technology, gain unauthorized access to a computer

system.

**attribute-based access control (ABAC):** A security mechanism that controls access to security resources by virtue of attributes that are attached to the resources and a set of rules defining combinations of attributes that identify the (dynamic) set of resources accessible to particular security principals, roles, or groups.

**availability zone:** A common feature of public clouds to provide a degree of separation and resiliency between different parts of the cloud platform. The cloud operator guarantees that, as a result of its platform design and operation, multiple availability zones are extremely unlikely to suffer failures at the same time. This provides cloud customers with options for moving workload from a failed availability zone to a surviving one, thereby increasing the resilience of their systems in the case of problems with the cloud platform.

**botnet:** A network of software **robots** (“bots”) that can be instructed to act in unison to perform predetermined tasks. The term is typically associated with large networks of bots programmed to perform offensive security actions, such as denial-of-service attacks.

**business benchmarking:** Determining common business trend across multiple institutions in a domain. It is a common practice for multitenant technology platforms. For example, for our TFX case study, a business benchmark could be the average delivery rates for small Asian companies versus European companies.

**cloud infrastructure:** Cloud computing refers to the practice of transitioning computer services such as computation or data storage to multiple, redundant offsite locations available on the Internet, which allows application software to be operated using Internet-enabled devices. Clouds can be classified as public, private, and hybrid.

**clustering:** A hardware and software technique for maximizing system availability by having a group of hardware or software entities cooperate in such a way that a failure in one of them is masked by one or more of the surviving entities taking over the workload. Variants can include a collocated cluster and a cluster with members that are geographically separated. The process of moving the work from the failed entity to a survivor is termed **failover**. The process of moving work back to a

recovered entity is termed **failback**. Clustering is normally an important mechanism for **high-availability** computing.

**connector:** In this context, an *architectural connector*, which is an architectural element used to connect two architectural components, such as a remote procedure call, a message channel, a file, an event bus, or any other communication channel for the purpose of transferring data or signals between them.

**container:** In this context, *software container*, which is a set of packaging and runtime conventions to provide highly efficient operating system virtualization and runtime isolation for a specific package of software and dependencies.

**continuous delivery:** Continuous delivery (CD) is a software engineering approach in which teams keep producing valuable software in short cycles and ensure that the software can be reliably released at any time. It is used in software development to automate and improve the process of software delivery. Techniques such as automated testing and continuous integration allow software to be developed to a high standard and easily packaged and deployed to test environments, resulting in the ability to rapidly, reliably and repeatedly push out enhancements and bug fixes to customers at low risk and with minimal manual overhead.

**continuous integration:** Continuous integration (CI) is the practice, in software engineering, of merging all developer working copies with a shared mainline several times a day. It was first named and proposed by Grady Booch in his 1991 method, although practices at this time did not yet support full automation or the performance of integrations more than a few times a day. It was adopted as part of Extreme Programming (XP), which did advocate integrating more than once per day, perhaps as many as tens of times per day. The main aim of CI is to prevent integration problems.

**continuous testing:** Continuous testing uses automated approaches to significantly improve the speed of testing by taking a so-called shift-left approach, which integrates the quality assurance and development phases. This approach may include a set of automated testing workflows, which can be combined with analytics and metrics to provide a clear, fact-based picture of the quality of the software being delivered.

**cryptography:** A security mechanism that protects information privacy and also enables other security mechanisms, such as data information integrity and nonrepudiation. The two fundamental operations are encryption and decryption. Encryption is an operation whereby source data (the **plain text**) is combined with a secret (or **key**) via an encryption algorithm to a form (the **cypher text**) from which the source data cannot be recovered in reasonable time without the secret. Decryption is the reverse of encryption, allowing the cypher text to be combined with a secret, via a reverse algorithm of the encryption algorithm, to recover the plain text. Two major variants of cryptography are *secret key* cryptography and *public/private key* (or just **public key**) cryptography. These two variants are also known as **symmetric key** and **asymmetric key** cryptography. Secret key cryptography is the process just described, using a single secret shared between the encrypting and decrypting actors. The weakness of secret key cryptography is the difficulty of passing the secret securely between the actors. Public key cryptography is a more complex, but widely applied, variant, which solves the problem of sharing secrets. Public key cryptography uses two mathematically related keys per principal, a public key and a private key. The public key encryption and decryption algorithms allow the plain text to be encrypted with the public key and decrypted with the private key. Therefore, public key cryptography solves the problem of sharing secrets, but unfortunately, the tradeoff is that it is orders of magnitude slower than secret key cryptography. Consequently, most practical protocols, such as Transport Layer Security (TLS), use both; public key cryptography is used to exchange a secret key, and then the secret key is used to encrypt the payload data that needs to be kept private.

**cryptographing hashing:** A security mechanism that allows changes in data to be detected. A cryptographic hashing algorithm maps data of arbitrary size, combined with a secret of some form, to a large numeric value (a **bit array**, known as the **hash**) and is infeasible to invert. This allows a large data item to be hashed, and then, at a later point, the data is rehashed and compared with the hash value to check whether the data has been changed.

**database replication:** An automated mechanism to continually copy all or some of the data in one database (the **primary**) to another database (the **replica**) with the minimum of delay between changes that are made in

the primary being applied to the replica. Typically, replication is an asynchronous operation (for performance reasons), and so, in the case of failure of the primary, there is normally some data loss between it and the replica.

**DBCC:** An abbreviation for Database Console Commands, a process of ensuring internal consistency of a database such as Microsoft SQL Server and in that product, provided by a command called `DBCC CHECKDB`.

**denial of service:** A form of security attack whereby the attacker's aim is to prevent normal operation of the system by compromising the system's availability rather than to compromise privacy, integrity, or other security properties. A denial-of-service attack often takes the form of a network attack that overloads the system by attempting to overwhelm it with a huge volume of requests.

**DevOps:** DevOps (a portmanteau of development and operations) is an application delivery philosophy that stresses communication, collaboration, and integration between software developers and their information technology (IT) counterparts in operations. DevOps is a response to the interdependence of software development and IT operations. It aims to help an organization rapidly produce software products and services.

**DevSecOps:** DevSecOps is an extension of the DevOps approach to also integrate security into the process, along with development and operations.

**disaster recovery (DR):** A set of policies, tools, and procedures to enable the recovery or continuation of vital technology infrastructure and systems following a natural or human-induced disaster.

**Domain-Driven Design (DDD):** A software design approach, pioneered by Eric Evans, that emphasizes that the structure and language of the software code should closely match the business domain. This allows the business people and software developers to have a shared and unambiguous language to use when discussing the problem to be solved (the so-called ubiquitous language).

**event:** An occurrence within a particular system or domain; it is something that has happened. The word **event** is also used to mean a programming entity that represents such an occurrence in a computing system.

**feature:** The Institute of Electrical and Electronics Engineers defines **feature** as “a distinguishing characteristic of a software item (e.g., performance, portability, or functionality).”<sup>3</sup>

<sup>3</sup> IEEE Computer Society, *IEEE Standard for Software and System Documentation* (Institute of Electrical and Electronics Engineers, 2008).

**high availability:** In this context, the technology used to reduce the impact of failures and maximize system availability using techniques outside the application software, such as hardware and software clusters, data replication, and failover and fallback mechanisms. (Contrast with **resilience**.)

**incident management:** The process of recognizing and managing the response to an unplanned interruption to a service or an unplanned reduction in a quality attribute of a service, leading to full service being restored.

**information technology (IT) organization:** Department within the enterprise that is in charge of creating, monitoring, and maintaining information technology systems, applications, and services.

**Information Technology Infrastructure Library (ITIL):** A service management framework that provides a set of detailed practices for IT service management (ITSM) that focuses on aligning IT services with the needs of business. It describes organization and technology independent processes and procedures to guide an organization toward strategy, service design, and basic competence in IT service delivery.

**injection attack:** A security attack on a piece of software that involves passing unexpected values in parameters that cause the software to malfunction in an insecure manner. The best-known form of injection attack is a **SQL injection attack**, but almost any interpreter in your software can be prone to this problem, leading to potential database injection attacks, configuration injection attacks, JavaScript injection attacks, and more.

**Kubernetes:** A piece of middleware, based on the design of Google’s Borg platform, that provides a runtime environment for software container execution, orchestration, and management.

**master (of data):** Saying that a particular component masters the data implies that only that component is allowed to store the value of that

data. Any other component that requires the value of the dataset can request it from the master. Other components can also request the master component to update the dataset; however, the master is responsible for applying relevant business rules required by the domain of interest. Applying such a pattern ensures that the dataset is always represented in a consistent manner across a system (or system of systems).

**metadata:** Data about data. Metadata represents data that provides information about other data, where the other data is normally the data that represents the domain. Examples of metadata include system-relevant information such as when the data was last updated and who updated it, security-relevant information such as security classification, and data provenance information such as source system and other lineage details.

**microservices:** In computing, microservices is a software architecture style in which complex applications are composed of small, independent processes communicating with each other using language-agnostic APIs. These services are small, highly decoupled, and focused on doing a small task.

**minimum viable product:** In product development, the minimum viable product (MVP) is the product with the highest return on investment versus risk. A minimum viable product has just those core features that allow the product to be deployed, and no more. The product is typically deployed to a subset of possible customers, such as early adopters that are thought to be more forgiving, more likely to give feedback, and able to grasp a product vision from an early prototype or marketing information. It is a strategy targeted at avoiding building products that customers do not want and that seeks to maximize the information learned about the customer per dollar spent.

**phishing attacks:** A kind of security attack that is a fraudulent attempt to obtain sensitive information or data, such as usernames and passwords by disguising the malicious actor as a trustworthy entity in an electronic communication. A common form of phishing is emails that appear to be from a trusted entity, such as an employer, payments provider, bank, or major software vendor, and that try to persuade the victim to click on a link in the email and then provide sensitive data through a Web form, mocked up to look as if it is created by the trusted entity.

**public key cryptography:** A form of **cryptography** that uses a pair of mathematically related keys, one that must be kept secret and one, known as the **public key**, that can be freely distributed. Encryption is performed with the public key and decryption with the private key.

**quality attribute:** A measurable or testable property of a system used to gauge the compliance of the system with the needs of its stakeholders.

**race condition:** Represents when a system's behavior is dependent on a sequence or timing of events. It can have multiple interpretations, but for data processing, a standard example is when a data-processing entity that is used by an independent component is updated by another component. As a result, the data processing is concluded with incorrect data because the original value of the data entity was changed.

**ransomware:** A type of malware that infects a computer and threatens to publish the victim's data or perpetually block access to it unless a ransom is paid to the attacker. A common sort of ransomware encrypts the files or even all of the storage devices that it finds on the machine.

**resilience:** In this context, approach to maximizing system availability by accepting the likelihood of failure and designing our software system to behave in a predictable way that reduces the impact of unexpected failures. (Contrast with **high availability**.)

**role-based access control (RBAC):** A security mechanism that allows access to security resources (e.g., files, actions, data items) to be controlled using roles that are assigned to security principals. Access control authorizations are allocated to roles, and security principals are assigned one or more roles to grant them the associated authorizations.

**RUP methodology:** The Rational Unified Process (RUP) is an iterative software development process framework created by the Rational Software Corporation, a division of IBM since 2003. RUP is not a single concrete prescriptive process but rather an adaptable process framework intended to be tailored by the development organizations and software project teams that will select the elements of the process that are appropriate for their needs. RUP is a specific implementation of the unified process.

**Scaled Agile Framework (SAFe):** "The Scaled Agile Framework (SAFe) is a freely revealed, online knowledge base of proven success patterns for applying Lean and Agile development at enterprise scale. The SAFe

website ([www.scaledagileframework.com](http://www.scaledagileframework.com)) allows users to browse the ‘SAFe Big Picture’ to understand the roles, teams, activities, and artifacts necessary to scale these practices to the enterprise.”<sup>4</sup>

<sup>4</sup> Ben Linders, “Lean and Agile Leadership with the Scaled Agile Framework (SAFe)” Interview with Dean Leffingwell, *InfoQ* (January 12, 2015). <http://www.infoq.com/news/2015/01/lean-agile-leadership-safe>

**schema:** A description of the structure of data in a system, including fields and datatypes. Commonly used for databases where it is expected that data stored in the database will comply with the defined schema.

**schema on read versus schema on write:** Architectural patterns for data integration, where schema on write implies that all data entered into a particular datastore complies with the schema before being stored. Schema on read assumes that data can be stored without such a restriction. The data in this case has to be converted to a schema required by the component that is accessing it.

**secret key cryptography:** A form of cryptography using a single secret key for encryption and decryption.

**service delivery management (SDM):** A set of standardized activities that are performed by an organization to design, plan, deliver, operate, and control information technology services offered to internal or external customers.

**single sign-on (SSO):** A security mechanism whereby a security principal (e.g., an end user or a network service) can authenticate itself once and then use a resulting security token of some form to prove its identity to a range of network services, thereby avoiding repeated authentication steps.

**software supply chain:** The set of sources from which all of the third-party software in a system is assembled. Today, this term is often used to refer to the large amount of open source software (and all of the third-party dependencies it requires) that are included in a system.

**SQL injection attack:** A security injection attack using parameters (typically in a Web request) that will be used to construct a SQL statement that is executed in the system’s database. With careful crafting, a parameter value that is then carelessly used to construct a SQL statement can often be used to completely change the meaning of the

SQL statement and reveal much more information than was intended or can even update the database.

**state machine:** A mathematical abstraction for a software component that operates by being in a particular state at any point in time and transitions between states according to the permitted transitions between the states, as defined by the state machine. Transitions are usually associated with a condition, based on the state of the software component, that defines whether or not the transition can be traversed to change state. Known as *statecharts* in the UML2 modeling language.

**system of engagement:** Information **technology** system that helps facilitate customer contacts with the enterprise by providing customized and smooth interactions.

**system of record:** Information **technology** system responsible for providing the authoritative source for data elements or information within the enterprise.

**technical debt:** (1) The complete set of technical debt items associated with a system. (2) In software-intensive systems, design or implementation constructs that are expedient in the short term but that set up a technical context that can make a future change more costly or impossible. Technical debt is a contingent liability **whose** impact is limited to internal system qualities, primarily, but not only, maintainability and evolvability.<sup>5</sup>

<sup>5</sup> Philippe Kruchten, Robert Nord, and Ipek Ozkaya, *Managing Technical Debt* (Software Engineering Institute/Carnegie Mellon, 2019).

**technical feature:** A piece of functionality that does not directly deliver business value but addresses some current or future technical issue with the system.

**two-factor authentication (2FA):** A security authentication mechanism that uses two different factors to identify the user, typically something that the user knows and something that the user has. A common form of 2FA is to use a password (that the user *knows*) and numeric values generated by an authenticator application (that the user *has*).

**user story:** A tool used in agile software development to capture a description of a software feature from an end-user perspective. The user

story describes the type of user, what the user wants, and why. A user story helps to create a simplified description of a requirement.