



Designing Software Architectures

A Practical Approach

SECOND EDITION



Early
Release
RAW & UNEDITED

Humberto Cervantes

Rick Kazman

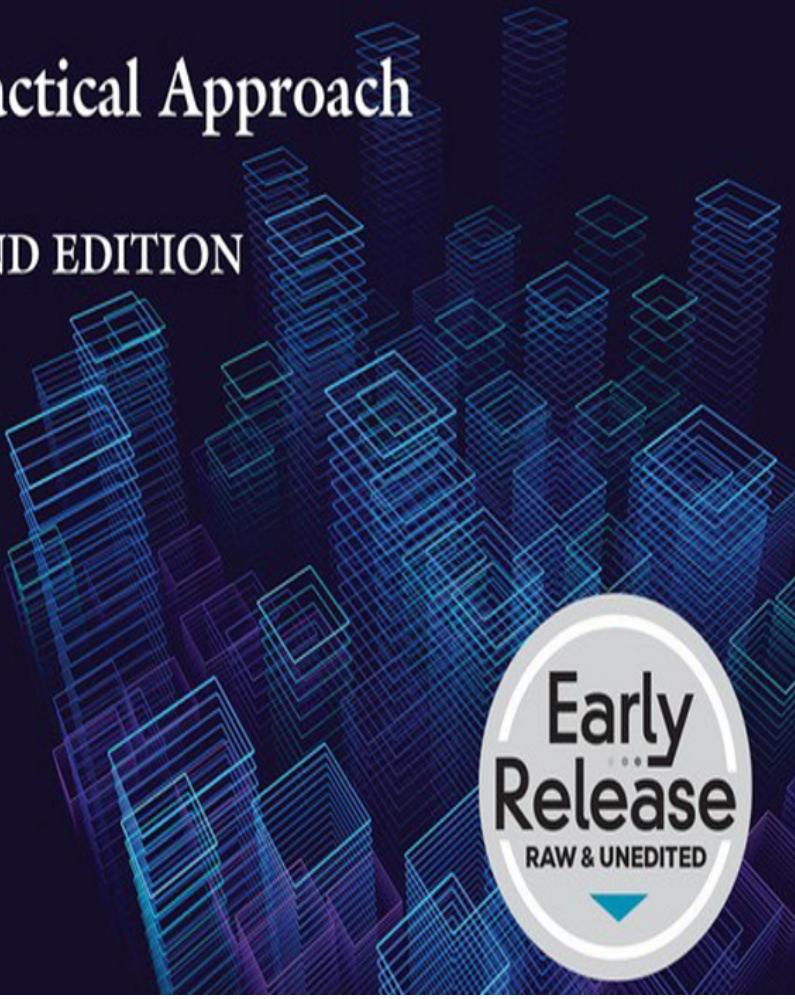


SEI SERIES IN SOFTWARE ENGINEERING

Designing Software Architectures

A Practical Approach

SECOND EDITION



Early
Release
RAW & UNEDITED



Humberto Cervantes

Rick Kazman

Designing Software Architectures: A Practical Approach, 2nd Edition

Humberto Cervantes and Rick Kazman

A NOTE FOR EARLY RELEASE READERS

With Early Release eBooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this title, please reach out to Pearson at PearsonITAcademics@pearson.com

Contents

Preface

Chapter 1. Introduction

Chapter 2. Architectural Design

Chapter 3. Making Design Decisions

Chapter 4. The Architecture Design Process

Chapter 5. Supporting Business Agility through API-Centric Design

Chapter 6. Designing for Deployability

Chapter 7. Designing Cloud-Based Solutions

Chapter 8. Hotel Pricing System: Case Study

Chapter 9. Digital Twins: Case Study

Chapter 10. Technical Debt in Design

Chapter 11. Analysis in the Design Process

Chapter 12. Architecture Design Process in the Organization

Chapter 13. Final Thoughts

Appendix A. Tactics-Based Questionnaires

Table of Contents

Preface

1. Introduction

 1.1 Motivations

 1.2 Software Architecture

 1.3 The Role of the Architect

 1.4 A Brief History of ADD

 1.5 Summary

 1.6 Further Reading

 1.7 Discussion Questions

2. Architectural Design

 2.1 Design in General

 2.2 Design in Software Architecture

 2.3 Why Is Architectural Design So Important?

 2.4 Architectural Drivers

 2.5 Summary

 2.6 Further Reading

 2.7 Discussion questions

3. Making Design Decisions

 3.1 Making Design Decisions

 3.2 Design Concepts: The Building Blocks for Creating Structures

- 3.3 Design Concepts to Support Performance
 - 3.4 Design Concepts to Support Availability
 - 3.5 Design Concepts to Support Modifiability
 - 3.6 Design Concepts to Support Security
 - 3.7 Design Concepts to Support Integrability
 - 3.8 Summary
 - 3.9 Further Reading
 - 3.10 Discussion Questions
- 4. The Architecture Design Process
 - 4.1 The Need for a Principled Method
 - 4.2 Attribute-Driven Design 3.0
 - 4.3 Applying ADD to different system contexts
 - 4.4 Identifying and Selecting Design Concepts
 - 4.5 Producing Structures
 - 4.6 Defining Interfaces
 - 4.7 Creating Preliminary Documentation During Design
 - 4.8 Tracking Design Progress
 - 4.9 Summary
 - 4.10 Further Reading
 - 4.11 Discussion Questions
 - 5. Supporting Business Agility through API-Centric Design
 - 6. Designing for Deployability
 - 6.1 Deployability Principles and Architectural Design

6.2 Design Decisions to Support Deployability

6.3 Deployability and ADD

6.4 Summary

6.5 Further Reading

6.6 Discussion Questions

7. Designing Cloud-Based Solutions

7.1 Introduction to the cloud

7.2 Drivers and the Cloud

7.3 Cloud-Based Design Concepts

7.4 ADD in the design of cloud-based solutions

7.5 Summary

7.6 Further Reading

7.7 Discussion Questions

8. Case Study - Hotel Pricing System

8.1 Business Case

8.2 System Requirements

8.3 Development and Operations Requirements

8.4 The Software Design Process

8.5 Summary

8.6 Further Reading

8.7 Discussion questions

9. Case Study – Digital Twin Platform

9.1 Business Case

9.2 System Requirements

9.3 The Design Process

9.4 Summary

9.5 Further Reading

10. Technical Debt in Architectural Design

10.1 Technical Debt

10.2 The Roots of Technical Debt in Design

10.3 Refactoring and Redesign

10.4 Technical Debt and ADD

10.5 Summary

10.6 Further Reading

10.7 Discussion Questions

11. Analysis in the Design Process

11.1 Analysis and design

11.2 Why Analyze?

11.3 Analysis Techniques

11.4 Tactics-based Analysis

11.5 Reflective Questions

11.6 Scenario-Based Design Reviews

11.7 Summary

11.8 Further Reading

11.9 Discussion questions

12. The Architecture Design Process in the Organization

12.1 Architecture Design and the Development Life Cycle

12.2 Architecture Design and the Organization

12.3 Summary

12.4 Further Reading

12.5 Discussion questions

13. Final Thoughts

13.1 On the Need for Methods

13.2 Future Directions

13.3 Next Steps

13.4 Further Reading

13.5 Discussion questions

Appendix A. Tactics-Based Questionnaires

A.1 Availability

A.2 Deployability

A.3 Energy Efficiency

A.4 Integrability

A.5 Modifiability

A.6 Performance

A.7 Safety

A.8 Security

A.9 Testability

A.10 Usability

A.11 Further Reading

Preface

It has now been nearly 8 years since the first edition of *Designing Software Architectures* appeared. Much has changed in the world of technology since then. Cloud architectures, IoT architectures, DevOps, the rise of AI/ML, containers, micro-services, and much more. Was our advice from 8 years ago still relevant? Well, yes and no.

The good news, from our perspective, is that the principles and practices of designing software architectures have not changed in 8 years. The ADD method, which provides the scaffolding for the entire book, did not change *at all*. During this time it has been taught to thousands of practitioners and used successfully in many industrial projects. In all this time, the feedback has been positive and we have not had requests to change the method. That was reassuring. What did change, though, was the technical environment and the contexts in which we design today.

It is rare today that someone designs a standalone system. At the least you will be building on top of existing frameworks and toolkits, incorporating off-the-shelf components, some of which are likely open source. But you may be building a system that is interacting with other systems in real time, possibly sharing resources with them. It is likely that you are building a system using some kind of Agile methodology and it is also likely that your system will be frequently modified, with regular updates and releases. It is possible that the architectures that you build will include IoT devices, will be mobile, will be containerized, and will be adaptive. And it is also possible that the software you are

working with is old, and has accumulated technical debt over the years.

For these reasons, we felt that it was time to write a second edition, one that acknowledges the many new contexts in which architectural decisions are made today. In this new edition we now have chapters on Supporting Business Agility through API-centric Design, on Deployability, on Cloud-based Solutions, and on Technical Debt in Design.

We also created two new case studies for this edition of the book, that build upon and explore the challenges in these new contexts. The Hotel Pricing System case study implements much of its functionality in microservices, is deployed to a cloud infrastructure, is meant to be portable across environments, and was designed to be highly available.

The other case study--a Digital Twin platform--explored even more challenges: IoT, cloud computing, big data and analytics, AI/ML, Extended Reality (XR), simulation, advanced automation, and often robotics. This was a very large, complex system and required the coordinated efforts of not just a single architect, but a large architecture team with varied expertise.

In each case we show how the ADD method helped to transform design challenges into reality in a disciplined way. We sincerely hope that this book gives you the confidence you need to tackle any design challenge, no matter how large!

1. Introduction

In this chapter we provide an introduction to the topic of software architecture and architecture design. We briefly discuss what architecture is and why it is fundamental to take it into account when developing software systems. We also discuss the activities that are associated with the development of software architecture so that architectural design—which is the primary topic of this book—can be understood in the context of these activities. We also briefly discuss the role of the architect, who is the person responsible for creating the design. Finally, we introduce the Attribute-Driven Design (ADD) method, the architecture design method that we will discuss extensively in this book.

1.1 Motivations

Our goal in this book is to teach you how to design software architecture in a systematic, predictable, repeatable, and cost-effective way. If you are reading this book, then presumably you already have an interest in architecture and aspire to be an architect. The good news is that this goal is within your grasp. To convince you of that point, we will spend a few moments talking about the idea of design—the design of anything—and we will see how and why architectural design is not so different. In most fields,

“design” involves the same sorts of challenges and considerations—meeting stakeholder needs, adhering to budgets and schedules, dealing with constraints, working with available materials, and so forth. While the primitives and tools of design may vary from field to field, the goals and steps of design do not.

This is encouraging news, because it means that design is not the sole province of wizards or highly skilled craftsmen. That is, design can be taught, and it can be learned. Most design, particularly in engineering, consists of putting known design primitives together in (sometimes innovative) ways that achieve predictable outcomes. Of course, the devil is in the details, but that is why we have methods. It may seem difficult at first to imagine that a creative endeavor such as design can be captured in a step-by-step method; this, however, is not only possible but also valuable, as Parnas and Clements have discussed in their paper *A Rational Design Process: How and Why to Fake It*. Of course, not everyone can be a great designer, just as not everyone can be a Thomas Edison or a LeBron James or a Frank Gehry. What we do claim is that everyone can be a *much better* designer, and that structured methods supported by reusable chunks of design knowledge, which we provide in this book, can help pave the road from mediocrity to excellence and from a craft to a discipline of engineering.

Our goal in writing this book is to provide a practical method that can be enacted by any competent software engineer, and also (and just as important) to provide a set of rich case studies that realize the method. Albert Einstein was reputed to have said, “Example isn’t another way to teach, it is the only way to teach.” We firmly believe that. Most of us learn better from examples than from sets of rules or steps or principles. Of course, we need the steps and rules and

principles to structure what we do and to create the examples, but the examples speak to our day-to-day concerns and help us by making the steps concrete. Experienced architects may not need to follow these steps precisely, but it is important to have them detailed so that novice architects can be guided in the design process.

This is not to say that architecture design will ever be simple. If you are building a complex system, the chances are that you are trying to balance many competing forces—things like time to market, cost, performance, evolvability, usability, availability, security, and so on. If you are pushing the boundaries in any of these dimensions, then your job as an architect will be even more complex. This is true in any engineering discipline, not just software. The groundbreaking Sydney Opera House was completed 10 years late and nearly 14 times over its original budget! If you examine the history of building large ships or skyscrapers or any other complex “system,” you will see how the architects of those systems struggled with making the appropriate decisions and tradeoffs. No, architecture design may never be easy, but our purpose is to make it tractable and achievable by well-trained, well-educated software engineers.

1.2 Software Architecture

Much has been written on what *software architecture* is. We adopt the definition of software architecture from *Software Architecture in Practice* (fourth edition):

The software architecture of a system is the set of structures needed to reason about the system. These structures comprise software elements, relations among them, and properties of both.

As you will see, our design method embodies this definition and helps to guide the designer in creating an architecture that has the *desired* properties.

1.2.1 The Importance of Software Architecture

Much has also been written on why *architecture* is important. Again, following *Software Architecture in Practice*, we note that architecture is important for a wide variety of reasons, and a similarly wide variety of consequences stem from those reasons:

- An architecture will inhibit or enable a system's driving quality attributes.
- The decisions made in an architecture allow you to reason about and manage change as the system evolves.
- The analysis of an architecture enables early prediction of a system's qualities.
- A documented architecture enhances communication among stakeholders.
- The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.
- An architecture defines a set of constraints on subsequent implementation.
- The architecture influences the structure of an organization, and vice versa.
- An architecture can provide the basis for evolutionary, or even throwaway, prototyping.
- An architecture is the key artifact that allows the architect and the project manager to reason about cost

and schedule.

- An architecture can be created as a transferable, reusable model that forms the heart of a product line.
- Architecture-centric development focuses attention on the assembly of components, rather than simply on their creation.
- By restricting design alternatives, architecture channels the creativity of developers, reducing design and system complexity.
- An architecture can be the foundation for training a new team member.

If an architecture is important for all of these reasons—if it affects the structure of the organization, and the qualities of the system, and the people involved in its creation and evolution—then surely great care must be taken in designing this crucial artifact. Sadly, that is most often not the case. Architectures often “evolve” or “emerge.” While evolution is typically a given and while emergence might be necessary as requirements congeal, and while we emphatically are not arguing for “big design up front,” doing no architecture at all is often too risky for anything but the simplest projects or products. Would you want to drive over a bridge or ride in a jet that had not been carefully designed? Of course not. But you use software every day that is buggy, costly, insecure, unreliable, fault prone, and slow—and many of these undesirable characteristics can be easily avoided!

The core message of this book is that architecture design does not need to be difficult or scary; it is not the sole province of wizards; and it does not have to be costly and all done up front. Our job is to show you how and convince you that it is within your reach.

1.2.2 Life-Cycle Activities

Software architecture design is one of the software architecture life-cycle activities ([Figure 1.1](#)). As in any software project life cycle, this activity is concerned with the translation of requirements into a design and from there into an implementation. Specifically, the architect needs to worry about the following aspects:

- *Architectural requirements.* Among all the requirements, a few will have a particular importance with respect to the software architecture. These *architecturally significant requirements (ASRs)* include not only the most important functionality of the system and the constraints that need to be taken into account, but also—and most importantly—quality attributes such as high performance, high availability, ease of evolution, and iron-clad security. These requirements, along with a clear design purpose and other architectural concerns that may never be written down or may be invisible to external stakeholders, will guide you to choose one set of architectural structures and components over another. We will refer to these ASRs, constraints, and concerns as *drivers*, as they can be said to drive the design.
- *Architectural design.* Design is a translation, from the world of needs (requirements) to the world of solutions, in terms of structures composed of modules, frameworks, and components. A good design is one that *satisfies* the drivers. This topic is the focus of the book.
- *Architectural documentation.* Some level of preliminary documentation (or *sketches*) of the structures should be created as part of architectural design. This activity, however, refers to the creation of a more formal document from these sketches. If the project or product

is small and is well-understood, then architecture documentation may be minimal. In contrast, if the project is large, if distributed teams are collaborating, if it is expected to survive for a long time, or if significant technical challenges exist, then architectural documentation will repay the effort invested in this activity. While documentation is often avoided and derided by programmers, it is a standard, non-negotiable deliverable in almost every other engineering discipline. If your system is big enough and if it is mission critical, it should be documented. In other engineering disciplines, a “blueprint”—some sort of documented design—is an absolutely essential step in moving toward implementation and the commitment of resources.

- *Architectural evaluation.* As with documentation, if your project is nontrivial, then you owe it to yourself and to your stakeholders to evaluate it—that is, to ensure that the architectural decisions that have been made are appropriate to address the critical requirements. Would you deliver code without testing it? Of course not. Similarly, why would you commit enormous resources to fleshing out an architecture if you had not first “tested” the design? You might want to do this when first creating the system, or evolving it, or when putting it through a major refactoring. Most commonly evaluation is done informally and internally, but for truly important projects it is advisable to have a formal evaluation done by an external team.
- *Architectural implementation/conformance checking.* Finally, you need to implement the architecture that you have created (and evaluated). As an architect, you may need to tweak the design as the system grows and as requirements emerge and evolve. This is normal. In addition to this tweaking, your major responsibility

during implementation is to ensure conformance of the code to the design. If developers are not faithfully implementing the architecture, they may be undermining the qualities that you have carefully designed in. Again, consider what is done in other fields of engineering. When a concrete foundation for a new building is poured, the building that rests on top of that foundation is not constructed until the foundation has first been tested, typically via a core sample, to ensure that it is strong enough, dense enough, sufficiently impermeable to water and gasses, and so forth. Without conformance checking, we have no way of ensuring the quality of what is being subsequently constructed.

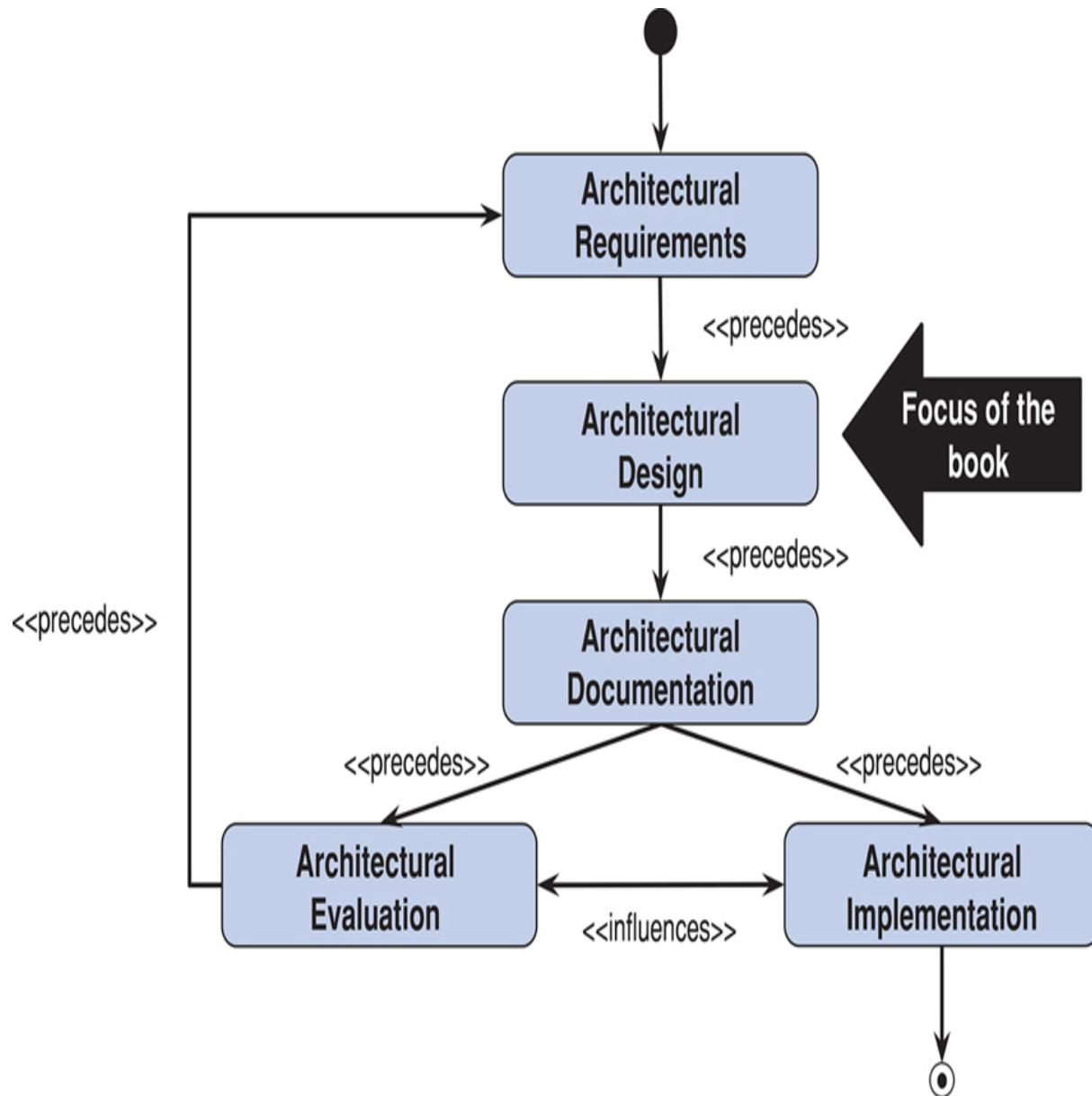


FIGURE 1.1 Software architecture life-cycle activities

Note that we are not proposing a specific life-cycle model in [Figure 1.1](#). The stereotype `<<precedes>>` simply means that some effort in an activity must be performed, and hence precede, effort in a later activity. For example, you cannot (or at least should not) perform design activities if you have no idea about the requirements, and you cannot evaluate an architecture if you have not first made some design decisions.

Today most commercial software is developed using some form of Agile methodology. None of these architecture activities is incompatible with Agile principles and practices. The question for a software architect is not “Should I do Agile or architecture?”, but rather “How much architecture should I do up front versus how much should I defer until the project’s requirements have solidified somewhat?” and “How much of the architecture should I formally document, and when?” Agile and architecture are happy companions for many software projects.

We will discuss the relationship between architecture design and business agility in [Chapter 5](#). And we will discuss the place of architecture design in an organizational context in [Chapter 12](#).

1.3 The Role of the Architect

An architect is much more than “just” a designer. This role, which may be played by one or more individuals, has a long list of duties, skills, and knowledge that must be satisfied if it is to be successful. These prerequisites include the following:

- *Leadership*: mentoring, team-building, establishing a vision, coaching
- *Communication*: both technical and nontechnical, encouraging collaboration
- *Negotiation*: dealing with internal and external stakeholders and their conflicting needs and expectations
- *Technical skills*: life-cycle skills, expertise with technologies, continuous learning, coding

- *Project skills*: budgeting, personnel, schedule management, risk management (although these duties are often shared with a project manager)
- *Analytical skills*: architectural analysis, general analysis mindset for project management and measurement (see the sidebar “[The Meaning of Analysis](#)”)

A successful design is not a static document that is “thrown over the wall.” That is, architects must not only design well, but must also be intimately involved in every aspect of the product or project, from conception and business justification to design and creation, through to operation, maintenance, and eventually retirement.

The Meaning of Analysis

In the *Merriam-Webster Dictionary*, the word *analysis* is defined as follows:

- The careful study of something to learn about its parts, what they do, and how they are related to each other
- An explanation of the nature and meaning of something

In this book we use the word *analysis* for different purposes, and both of these definitions apply. For instance, as part of the architectural evaluation activity, an existing architecture is analyzed to gauge if it is appropriate to satisfy its associated drivers. During the design process, the inputs are analyzed to make design decisions. The creation of prototypes is also a form of analysis. In fact, analysis is so important to the design process that we devote [Chapter 11](#) to just this topic.

In this book, we focus primarily on the design activity, its associated technical skills, and its integration into the development life-cycle. We will discuss the many roles of

the architect in [chapter 12](#) but for a fuller treatment of the other aspects of an architect’s life, we invite you to read a more general book on software architecture, such as *Software Architecture in Practice*.

1.4 A Brief History of ADD

While an architect has many duties and responsibilities, in this book we focus on what is arguably the single most important skill that a software engineer must master to be called “architect”: the process of design. To make architectural design more tractable and repeatable, in this book we focus most of our attention on the Attribute-Driven Design (ADD) method, which provides step-by-step guidance on how to iteratively perform the design activity shown in [Figure 1.1](#). [Chapter 4](#) describes the most recent version of ADD, version 3.0, in detail, so here we provide a bit of background for those who are familiar with previous versions of ADD. The first version of ADD (ADD 1.0, originally called ABD, for “Architecture-Based Design”) was published in January 2000, and the second version (ADD 2.0) was published in November 2006. Version 3.0 was published in 2016, in the first edition of this book.

ADD is, to our knowledge, the most comprehensive and most widely used documented architecture design method. When ADD appeared, it was the first design method to focus specifically on quality attributes and their achievement through the creation of architectural structures and their representation through views. Another important contribution of ADD is that it includes architecture analysis and documentation as an integral part of the design process. In ADD, design activities include refining the sketches created during early design iterations to produce a more detailed architecture, and continuously evaluating the design.

While ADD 2.0 was useful for linking quality attributes to design choices, it had several shortcomings that needed to be addressed: for example, it did not clearly relate patterns and tactics to implementation technologies (such as components and frameworks), it did not offer architects guidance for how to weave together architectural design and Agile practices, and it gave little indication as to how to begin the design process.

ADD 3.0 addresses these shortcomings. To be sure, ADD 3.0 is evolutionary, not revolutionary. It was catalyzed by the creation of ADD 2.5,¹ which was itself a reaction to attempting to use ADD in the real world, in many different contexts.

¹ This is our own coding notation; the 2.5 number is not used elsewhere.

We published ADD 2.5 in 2013. In that work, we advocated the use of application frameworks such as Spring, or Hibernate as first-class design concepts. This change was intended to address ADD 2.0's shortcoming of being too abstract to apply easily. ADD starts with drivers, systematically links them to design decisions, and then links those decisions to the available implementation options, including externally developed components. For Agile development, ADD 3.0 promotes quick design iterations in which a small number of design decisions are made, potentially followed by an implementation effort. In addition, ADD 3.0 explicitly promotes the (re)use of reference architectures and promotes the use of a “design concepts catalog,” which includes a broad selection of tactics, patterns, frameworks, reference architectures, and technologies.

Since the first version of this book, the field of architectural design has evolved again. During this time ADD 3.0 has been taught and used extensively in industry. Throughout

this experience we have not found any compelling reason to change it. This is a good thing: methods should be general and stable.

And so, while we are not updating the steps of the method itself, we felt the need to update the first edition of the book to reflect the changes surrounding ADD. What has changed are the ways that ADD can be used with contemporary architectural practices. Architects designing systems today are focusing on aspects such as Cloud, DevOps and Technical Debt. And we have also placed an emphasis in this edition on the design of APIs for distributed systems.

1.5 Summary

Having covered our motivations and background, we now move on to the heart and soul of this book. In the next few chapters, we describe what we mean by design and by architectural design in particular, we discuss ADD, and we provide two case studies showing in detail how ADD can be used in the real world. We also discuss the use of ADD in contemporary practices including cloud development, API-centric design agility and DevOps. We will also discuss the critical role that analysis plays in the design process and provide examples of how analysis can be performed on design artifacts and how the design process fits in an organizational context.

1.6 Further Reading

Fred Brooks has written a thoughtful series of essays on the nature of design, reflecting his 50 years of experience as a designer and researcher: F. P. Brooks, Jr. *The Design of Design: Essays from a Computer Scientist*. Addison-Wesley, 2010.

The usefulness of having a documented process for design and other development activities has been recognized for decades. This is discussed in D. Parnas and P. Clements, "A Rational Design Process: How and Why to Fake It," *IEEE Transactions on Software Engineering*, SE-12, 2, February 1986.

The definition of software architecture used here, as well as the arguments for the importance of architecture and the role of the architect, all derive from L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 4th ed., Addison-Wesley, 2021.

An early reference for the first version of ADD can be found in F. Bachmann, L. Bass, G. Chastek, P. Donohoe, and F. Peruzzi, *The Architecture Based Design Method*, CMU/SEI-2000-TR-001. The second version of ADD was described in R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord, and W. Wood, *Attribute-Driven Design (ADD), Version 2.0*, CMU/SEI-2006-TR-023. The version of ADD that we have referred to here as ADD 2.5 was published in H. Cervantes, P. Velasco-Elizondo, and R. Kazman, "A Principled Way of Using Frameworks in Architectural Design", *IEEE Software*, 46-53, March/April 2013.

1.7 Discussion Questions

1. Consider the architecture of buildings and software architecture. What are similarities and differences between these disciplines in terms of how they elicit requirements, the quality attributes they focus on, and how they make design decisions?
2. If you were to design the architecture of a software system, what steps would you follow? Would these steps change based on the complexity of that system?

3. Consider a system that is built by a team of developers, but none of them has knowledge of architectural design. Nonetheless they build the system the best they can. Does this system have an architecture?
4. Consider the definition of a software architecture given in [section 1.4](#). What type of elements and relations can form the structures of a software system?

2. Architectural Design

We now dive into a discussion of this complex and much misunderstood topic, architecture design. We will spend time talking about what it is, why it is important, how it works (at an abstract level), and which major concepts and activities it involves. We will also discuss architectural drivers: the various factors that “drive” design decisions, some of which may be documented as requirements, but many of which will not be.

Why read this chapter?

Design is the core concept of this book. Designs can be arbitrarily complex but you can't possibly deal with all of the complexity all of the time. This chapter gives you a way to think about design decisions at different levels of abstraction, and this is key to taming the complexity. Also, you may not intuitively consider all dimensions of drivers. Here we give you a framework for thinking about drivers and you will be using this framework throughout the rest of the book.

2.1 Design in General

Design is both a verb and a noun. Design is a process, an activity, and hence a verb. The process results in the creation of a design—a description of a desired state. The output of the design process is the thing, the noun, the artifact that you will eventually implement. Designing means making decisions to achieve goals and to satisfy requirements and constraints. The outputs of the design process are a direct reflection of those goals, requirements, and constraints. Think about houses, for example. Why do traditional houses in China look different from those in Switzerland or Algeria? Why does a yurt look like a yurt, which is different from an igloo or a chalet or a longhouse?

The architectures of these styles of houses have evolved over the centuries to reflect their unique sets of goals, requirements, and constraints. Houses in China feature symmetric enclosures, sky wells to increase ventilation, south-facing courtyards to collect sunlight and provide protection from cold north winds, and so forth. A-frame houses have steep pitched roofs that extend to the ground, meaning minimal painting and protection from heavy snow loads (which just slide off to the ground). Igloos are built of ice, reflecting the availability of ice, the relative poverty of other building materials, and the constraints of time (a small one can be built in an hour).

In each case, the process of design involved the selection and adaptation of a number of solution approaches. Even igloo designs can vary. Some are small and meant for a temporary travel shelter. Others are large, often connecting several structures, meant for entire communities to meet. Some are simple unadorned snow huts. Others are lined with furs, with ice “windows,” and doors made of animal skin.

The process of design, in each case, balances the various “forces” facing the designer. Some designs require

considerable skill to execute (such as carving and stacking snow blocks in such a way that they produce a self-supporting dome). Others require relatively little skill—a simple lean-to can be constructed from branches and bark by almost anyone. But the qualities that these structures exhibit may also vary considerably. Lean-tos provide little protection from the elements and are easily destroyed by high winds or fire, whereas an igloo can withstand Arctic storms and support the weight of a person standing on the roof.

Is design “hard”? Well, yes and no. *Novel* design is hard. It is pretty clear how to design a conventional bicycle, but the designs for the Segway and the OneWheel broke new ground. Fortunately, most design is not novel, because most of the time our requirements are not novel. Most people want a bicycle that will reliably convey them from place to place. Most people living in Phoenix want a house that can be easily and economically kept cool, whereas most people in Edmonton are primarily concerned with a house that can be kept warm. In contrast, people living in Tokyo and Mexico City are often concerned with buildings that can withstand serious earthquakes.

The good news for you, the architect, is that there are ample proven designs and design fragments, or building blocks that we call *design concepts*, that can be reused and combined to reliably achieve these goals. If your design is truly novel—if you are designing the next Sydney Opera House—then the design process will likely be “hard.” The Sydney Opera House, for example, cost nearly 14 times its original budget estimate and was delivered 10 years late. So, too, with the design of software architectures.

2.2 Design in Software Architecture

Architectural design for software systems is no different than design in general: It involves making decisions, working with available skills and materials, to satisfy requirements and constraints. In architectural design, we make decisions to transform our design purpose, primary functional requirements, quality attribute requirements, constraints, and architectural concerns—what we call the architectural *drivers*—into structures, as shown in [Figure 2.1](#). These structures, which are mentioned in the definition of software architecture that we gave in [chapter 1](#), are then used to guide the project. They guide analysis and construction, and serve as the foundation for educating a new project member. They also guide cost and schedule estimation, team formation, risk analysis and mitigation, and, of course, implementation.

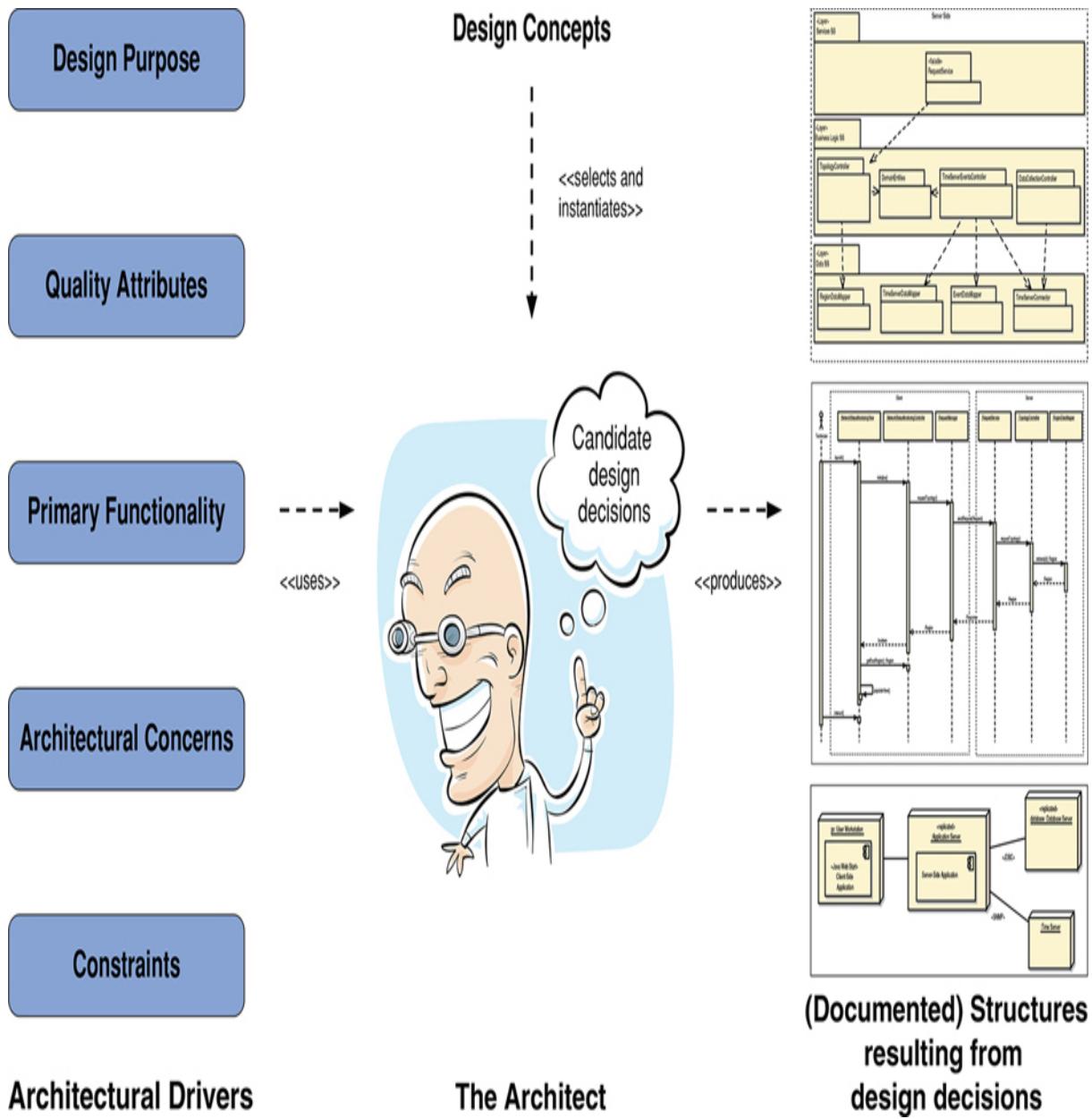


FIGURE 2.1 Overview of the architecture design activity

Architectural design is, therefore, a key step to achieving your product and project goals. Some of these goals are technical (e.g., achieving low and predictable latency in a video game or an e-commerce website), and some are nontechnical (e.g., supporting multiple lines of business, keeping your existing team employed, entering a new market, meeting a deadline). The decisions that you, as an

architect, make will have implications for the achievement of these goals and may, in some cases, be in conflict. The choice of a particular reference architecture (e.g., the Lambda Architecture) may provide a good foundation for achieving your latency and throughput goals and will keep your workforce employed because they are already familiar with that reference architecture and its supporting technology stacks. But this choice will not help you enter a new market—mobile games, for example.

In general, when designing, a change in some structure to achieve one quality attribute will have negative effects on other quality attributes. These *tradeoffs* are a fact of life for every practicing architect in every domain. We will see this over and over again in the examples and case studies provided in this book. Thus the architect's job is not one of finding an optimal solution, but rather one of *satisficing*—searching through a potentially large space of design alternatives and decisions until an acceptable solution is found.

2.2.1 Architectural Design

Grady Booch has said, “All architecture is design, but not all design is architecture.” What makes a decision “architectural”? A decision is architectural if it has non-local consequences *and* those consequences matter to the achievement of an architectural driver. No decision is, therefore, inherently architectural or non-architectural. The choice of a buffering strategy within a single element may have little effect on the rest of the system, in which case it is an implementation detail that is of no concern to anyone except the implementer or maintainer of that element. In contrast, the buffering strategy may have enormous implications for performance (if the buffering affects the achievement of latency or throughput or jitter goals) or

availability (if the buffers might not be large enough and information gets lost) or modifiability (if we wish to flexibly change the buffering strategy in different deployments or contexts). The choice of a buffering strategy, like most design choices, is neither inherently architectural nor inherently non-architectural. Instead, this distinction is completely dependent on the current and anticipated architectural drivers.

2.2.2 Element Interaction Design

Architectural design identifies and defines the major elements and important relationships that make up the overall structure of a system. The major elements are often abstract or existing design concepts (such as architectural patterns) that address primary functional requirements, quality attribute requirements, constraints, and architectural concerns. Architectural design generally results in the identification of only a subset of the elements that are part of the system's structure. This is to be expected because, during initial architectural design, the architect will focus on the primary functionality of the system.

What makes a use case primary? A combination of business importance, risk, and complexity considerations feed into this designation. Of course, to your typical user almost everything will be urgent and number one priority. More realistically, a small number of user stories provide the most fundamental business value or represent the greatest risk (if they are done wrong), so these are deemed primary. Every system has many more user stories, beyond the primary ones, that need to be satisfied. The elements that support these nonprimary user stories and their interfaces are identified as part of what we call *element interaction design*. This level of design usually follows architectural design. The location and relationships of these elements, however, are

constrained by the decisions that were made during architectural design. These elements can be units of work (i.e., modules or services) assigned to an individual or to a team, so this level of design is important for defining not only how nonprimary functionality is allocated, but also for planning purposes (e.g., team formation and communication, budgeting, outsourcing, release planning, unit and integration test planning).

Depending on the scale and complexity of the system, the architect should be involved in element interaction design, either directly or in an auditing role. This involvement ensures that the system's important quality attributes are not compromised—for example, if the elements are not defined, located, and connected correctly. It will also help the architect spot opportunities for generalization.

2.2.3 Element Internals Design

A third level of design follows element interaction design, which we call *element internals design*. In this level of design, which is usually conducted as part of the element development activities, the internals of the elements identified in the previous design level are established, so as to satisfy the element's interface.

2.2.4 Decisions and the design levels

Architectural decisions can and do occur at the three levels of design. Moreover, during architectural design, the architect may need to delve as deeply as element internals design to achieve a particular architectural driver. An example of this is the design of the internals of a microservice following a layered architecture. The data layer of the microservice may need to be carefully designed so that it does not contribute excessive latency or lock

important parts of its database. In this sense, architectural design can, at times, involve considerable detail, which explains why we do not like to talk about it in terms of “high-level design” or “detailed design” (see the sidebar “High Level vs. Detailed Design?”).

Architectural design precedes element interaction design, which precedes element internals design. This is logically necessary: One cannot design an element’s internals until the elements themselves have been defined, and one cannot usefully reason about interaction until several elements and some patterns of interactions among them have been defined. But as projects or products grow and evolve, there is, in practice, considerable iteration between these activities.

High level vs. Detailed Design?

The term “detailed design” is often used to refer to the design of the internals of modules. Although it is widely used, we really don’t like this term, which is presented as somehow in opposition to “high-level design.” We prefer the more precise terms “architectural design,” “element interaction design,” and “element internals design.”

After all, architectural design may be quite detailed, if your system is complex. And some design “details” will turn out to be architectural. For the same reason, we also don’t like the terms “high-level design” and “low-level design.” Who can really know what these terms actually mean? Clearly, “high-level design” should be somehow “higher” or more abstract, and cover more of the architectural landscape than “low-level design,” but beyond that we are at a loss to imbue these terms with any precise meaning.

So here is what we recommend: Just avoid using terms such as “high,” “low,” or “detailed” altogether. There is always a

better, more precise choice, such as “architectural,” “element interaction,” or “element internals” design!

Think carefully about the impact of the decisions you are making, the information that you are trying to convey in your design documentation, and the likely audience for that information, and then give that process an appropriate, meaningful name.

2.3 Why Is Architectural Design So Important?

While premature commitment can be wasteful, and sometimes “you ain’t gonna need it”, there is a very high cost to a project of *not* making important design decisions, or of not making them early enough. This manifests itself in many different ways. Early on, an initial architecture is critical for estimation in project proposals. Without doing some architectural thinking and some early design work, you cannot confidently predict project cost, schedule, and quality. Even at this early stage, an initial architecture will determine the key approaches for achieving architectural drivers, the gross work-breakdown structure, and the choices of tools, skills, and technologies needed to realize the system.

In addition, architecture is a key enabler of agility, as we will discuss in [Chapter 12](#). Whether your organization has embraced Agile processes or not, it is difficult to imagine anyone who would willingly choose an architecture that is brittle and hard to change or extend or tune—and yet it happens all the time. This so-called *technical debt* (which we discuss in [Chapter 10](#)) occurs for a variety of reasons, but paramount among these is the combination of a focus on features—typically driven by stakeholder demands—and the inability of architects and project managers to measure

the return on investment of good architectural practices. Features provide immediate benefit. Architectural improvement provides immediate costs and, if done right, long-term benefits. Put this way, why would anyone ever “invest” in architecture? The answer is simple: Without architecture, the benefits that the system is supposed to bring will be far harder to realize: velocity will slow and the system will become increasingly hard to debug, fix, scale, and evolve.

Simply put, if you do not make some key architectural decisions early and if you allow your architecture to degrade, you will be unable to maintain sprint velocity, because you cannot easily respond to change requests. In this light it should be clear that we vehemently disagree with what the original creators of the Agile Manifesto claimed: “The best architectures, requirements, and designs emerge from self-organizing teams.” Indeed, our demurral with this point is precisely why we have written this book. Good architectural design is difficult (and still rare), and it does not just “emerge.” This opinion mirrors a growing consensus within the Agile community. More and more, we see techniques such as “disciplined agility at scale,” the “walking skeleton,” and the “Scaled Agile Framework” embraced by Agile thought leaders and practitioners alike. Each of these techniques advocates some architectural thinking and design prior to much, if any, development. To reiterate, architecture enables agility, and not the other way around.

Furthermore, the architecture will influence, but not determine, other decisions that are not in and of themselves design decisions. These decisions do not influence the achievement of quality attributes directly, but they may still need to be made by the architect. For example, such

decisions may include selection of tools; structuring the development environment and making work assignments.

Finally, a well-designed, properly communicated architecture is key to achieving *agreements* that will guide the team. The most important kinds to make are agreements on interfaces and on shared resources.

Agreeing on interfaces early is important for component-based development, and critically important for distributed development using approaches such as microservices.

These decisions *will* be made sooner or later. If you don't make the decisions early, the system will be much more difficult to integrate. In [Section 4.6](#), we will discuss how to define interfaces as part of architectural design—both the external interfaces to other systems and the internal interfaces that mediate your element interactions.

2.4 Architectural Drivers

Before commencing design with ADD (or with any other design approach, for that matter), you need to think about what you are doing and why. While this statement may seem blindingly obvious the devil is, as usual, in the details. We categorize these “what” and “why” questions as architectural drivers. As shown in [Figure 2.1](#), these drivers include a design purpose, quality attributes, primary functional requirements, architectural concerns, and constraints. These considerations are critical to the success of the system and, as such, they *drive* and shape the architecture.

As with any other important requirements, architectural drivers need to be baselined and managed throughout the development life cycle.

2.4.1 Design Purpose

First, you need to be clear about the purpose of the design that you want to achieve. When and why are you doing this architecture design? Which business goals is the organization most concerned about at this time?

1. You may be doing architecture design as part of a project proposal (for the estimation process in a consulting organization, or for internal project selection and prioritization in a company, as discussed in [Section 12.1.1](#)). It is not uncommon that, as part of determining project feasibility, schedule, and budget, an initial architecture is created. Such an architecture would not be very detailed; its purpose is to understand and break down the architecture in sufficient detail that the units of work are understood and hence may be estimated.
2. You may be doing architecture design as part of the process of creating an exploratory prototype. In this case, the purpose of the architecture design process is not so much to create a releasable or reusable system, but rather to explore the domain, to explore new technology, to place something executable in front of a customer to elicit rapid feedback, or to explore some quality attribute (such as latency or performance scalability or failover for availability).
3. You may be designing your architecture during development. This could be for a complete new system, for a substantial portion of a new system, or for a portion of an existing system that is being refactored or replaced. In this case, the purpose is to do just enough design work to satisfy requirements, guide system construction and work assignments, and prepare for an eventual release.

In a mature domain the estimation process, for example, might be relatively straightforward; the architect can reuse existing systems as examples and confidently make estimates based on analogy. In novel domains, the estimation process will be far more complex and risky, and may have highly variable results. In these circumstances a prototype of the system, or a key part of the system, may need to be created to mitigate risk and reduce uncertainty. In many cases, this architecture may also need to be quickly adapted as new requirements are learned and embraced. In brownfield systems, where the requirements are better understood, the existing system is itself a complex artifact that must be well understood for planning to be accurate.

Finally, the development organization's goals during development or maintenance may affect the architecture design process. For example, the organization might be interested in designing for reuse, designing for future extension or subsetting, designing for scalability, designing a product for continuous delivery, designing to best utilize existing project capabilities and team member skills, and so forth. Or the organization might have a strategic relationship with a vendor. Or the CTO might have a specific technology preference and wants to impose it on your project.

Why do we bother to list these considerations? Because they *will* affect both the process of design and the outputs of design. Architectures exist to help achieve business goals. The architect should be clear about these goals and should communicate them (and negotiate them!) and establish a clear design purpose *before* beginning the design process.

2.4.2 Quality Attributes

In the book *Software Architecture in Practice*, *quality attributes* are defined as being measurable or testable properties of a system that are used to indicate how well the system satisfies the needs of its stakeholders. Because the term “quality”, as it is informally used, tends to be a squishy term, these properties allow quality to be expressed succinctly and objectively.

Among the drivers, quality attributes are the ones that shape the architecture the most. The critical choices that you make when you are doing architectural design determine, in large part, the ways that your system will or will not meet these driving quality attribute goals.

Given their importance, you must worry about eliciting, specifying, prioritizing, and validating quality attributes. Given that so much depends on getting these drivers right, this sounds like a daunting task. Fortunately, a number of well-understood, widely disseminated practices can help you here (see sidebar “[The Quality Attribute Workshop and the Utility Tree](#)” for a more detailed discussion about the two first techniques):

- A *Utility Tree*, the simplest technique, can be used by the architect to quickly prioritize quality attribute requirements according to their technical difficulty and risk.
- The *Quality Attribute Workshop (QAW)*, a bit more costly and complex, is a facilitated brainstorming session involving a group of system stakeholders that covers the bulk of the activities of eliciting, specifying, prioritizing, and achieving consensus on quality attributes.
- The *Mission Thread Workshop* serves the same purpose as QAW, but for a system of systems and it is typically the most costly, and most complex method. But if you

are building a system of systems, this is probably worth it.

The best way to discuss, document, and prioritize quality attribute requirements is as a set of scenarios. A *scenario*, in its most basic form, describes the system's response to some stimulus. Why are scenarios the best approach? Because all other approaches are worse! Endless time may be wasted in defining terms such as "performance" or "scalability" or "modifiability" or "configurability," as these discussions tend to shed little light on the real system. It is meaningless to say that a system will be "modifiable," because every system is modifiable with respect to some changes and not modifiable with respect to others. One can, however, specify the modifiability response measure you would like to achieve in response to a specific change request, perhaps in terms of elapsed time or total effort. For example, you might want to specify that "a request to update shipping rates is processed in less than 50 milliseconds"—an unambiguous criterion.

The heart of a quality attribute scenario, therefore, is the pairing of a stimulus with a response. Suppose that you are building a video game and you have a functional requirement like this: "The game shall change view modes when the user presses the <C> button." This functional requirement, if it is important, needs to be associated with quality attribute requirements. For example:

- How fast should the function be?
- How secure should the function be?
- How modifiable should the function be?

To address this problem, we use a *scenario* to describe a quality attribute requirement. A quality attribute scenario is a short description of how a system is required to respond to

some stimulus. For example, we might annotate the functional requirement given earlier as follows: “The game shall change view modes in < 500 ms when the user presses the <C> button.” A scenario associates a stimulus (in this case, the pressing of the <C> button) with a response (changing the view mode) that is measured using a response measure (< 500 ms). A complete quality attribute scenario adds three other parts: the source of the stimulus (in this case, the user), the artifact affected (in this case, because we are dealing with end-to-end latency, the artifact is the entire system) and the environment (are we in normal operation, startup, degraded mode, or some other mode?). In total, then, there are six parts of a completely well-specified scenario, as shown in [Figure 2.2](#).

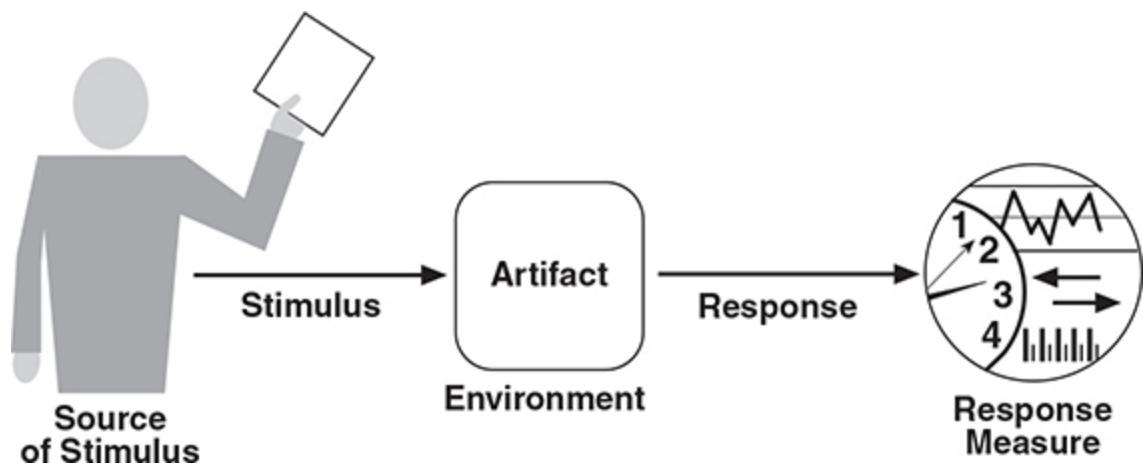


FIGURE 2.2 The six parts of a quality attribute scenario

Scenarios are testable, *falsifiable hypotheses* about the quality attribute behavior of the system under consideration. Because they have explicit stimuli and responses, we can evaluate a design in terms of how likely it is to support the scenario, and we can take measurements and test a prototype or fully fleshed-out system for whether it satisfies the scenario in practice. If the analysis (or prototyping results) indicates that the scenario’s response

goal likely won't be met, then the hypothesis is deemed to be falsified.

We often hear the objection "I don't know precisely what that requirement is" or "I don't know what that response measure should be". You should still collect and document scenarios, even in the face of uncertainty, for two reasons: 1) just struggling with this question, and perhaps discussing it, will help to clarify the requirement, and 2) even if precise requirements are not known, you probably know the value of the response measure to within an order of magnitude. If the system should respond "quickly" to a user's request for a report, you may not know whether 2 seconds or 5 seconds or 10 seconds is the best number, but you certainly know that 100 seconds doesn't count as "quickly" in most people's books. And this order-of-magnitude requirement is typically enough to guide architectural decisions.

As with all other requirements, scenarios should be prioritized. This can be achieved by considering two dimensions that are associated with each scenario:

- The first dimension corresponds to the importance of the scenario with respect to the success of the system. This is ranked by the customer.
- The second dimension corresponds to the degree of technical risk associated with the scenario. This is ranked by the architect.

A low/medium/high (L/M/H) scale is used to rank both dimensions; in practice we find that people don't have too much trouble making these coarse distinctions. Once the dimensions have been ranked, scenarios are prioritized by selecting those that have a (H, H) ranking or, if there aren't enough of those, we will look at scenarios with (H, M), or (M, H) rankings. Why these ones? If a scenario is high business

importance and high risk, this is definitely a driver! And it certainly deserves your attention.

In addition, some traditional requirements elicitation techniques can be modified slightly to focus on quality attribute requirements, such as Joint Requirements Planning (JRP), Joint Application Design (JAD), discovery prototyping, and user stories.

But whatever technique you use, *do not* start design without a prioritized list of measurable quality attributes! While stakeholders might plead ignorance (“I don’t know how fast it needs to be; just make it fast!”), you can always elicit at least a range of possible responses. Instead of accepting the requirement that the system should be “fast,” ask the stakeholder if a 10 second response time is acceptable. If that is unacceptable, ask if 5 seconds is OK, or, failing that, 1 second. You will find that, in most cases, users know more than they realize about their requirements, and you can at least “box them in” to a range.

The Quality Attribute Workshop and the Utility Tree

The Quality Attribute Workshop (QAW)

The QAW is a facilitated, stakeholder-focused method to generate, prioritize, and refine quality attribute scenarios. A QAW meeting is ideally enacted before the software architecture has been defined although, in practice, we have seen the QAW being used at all points in the software development life cycle. The QAW is focused on system-level concerns and specifically the role that software will play in the system. The steps of the QAW are as follows:

1. QAW Presentation and Introductions

The QAW facilitators describe the motivation for the QAW and explain each step of the method.

2. Business Goals Presentation

A stakeholder representing the project's business concerns presents the system's business context, broad functional requirements, constraints, and known quality attribute requirements. The quality attributes that will be refined in later QAW steps will be derived from, and should be traceable to, the business goals presented in this step. For this reason, these business goals must be prioritized.

3. Architectural Plan Presentation

The architect presents the system architectural plans as they currently exist. Although the architecture has frequently not been defined yet (particularly for greenfield systems), the architect often knows quite a lot about it even at this early stage. For example, the architect might already know about technologies that are mandated, other systems that this system must interact with, standards that must be followed, subsystems or components that could be reused, and so forth.

4. Identification of Architectural Drivers

The facilitators share their list of key architectural drivers that they assembled during steps 2 and 3 and ask the stakeholders for clarifications, additions, deletions, and corrections. The idea here is to reach a consensus on a distilled list of architectural drivers that covers major functional requirements, business drivers, constraints, and quality attributes.

5. Scenario Brainstorming

Given this context, each stakeholder now has the opportunity to express a scenario representing that stakeholder's needs and desires with respect to the system. The facilitators ensure that each scenario has an explicit stimulus and response. The facilitators also ensure traceability and completeness: At least one representative scenario should exist for each architectural driver listed in step 4 and should cover all the business goals listed in step 2.

6. Scenario Consolidation

Similar scenarios are consolidated where reasonable. In step 7, the stakeholders vote for their favorite scenarios, and consolidation helps to prevent votes from being spread across several scenarios that are expressing essentially the same concern.

7. Scenario Prioritization

Prioritization of the scenarios is accomplished by allocating to each stakeholder a number of votes equal to 30 percent of the total number of scenarios. The stakeholders can distribute these votes to any scenario or scenarios. Once all the stakeholders have voted, the results are tallied and the scenarios are sorted in order of popularity.

8. Scenario Refinement

The highest-priority scenarios are refined and elaborated. The facilitators help the stakeholders express these in the form of six-part scenarios: source, stimulus, artifact, environment, response, and response measure.

The output of the QAW is therefore a prioritized list of scenarios, aligned with business goals, where the highest-priority scenarios have been explored and refined. A QAW

can be conducted in as little as 2-3 hours for a simple system or as part of an iteration, and as much as 2 days for a complex system where requirements completeness is a goal.

Utility Tree

If no stakeholders are readily available to consult, you still need to decide what to do and how to prioritize the many challenges facing the system. One way to organize your thoughts is to create a Utility Tree. The Utility Tree, such as the one shown in [figure 2.3](#), helps to articulate your quality attribute goals in detail, and then to prioritize them.

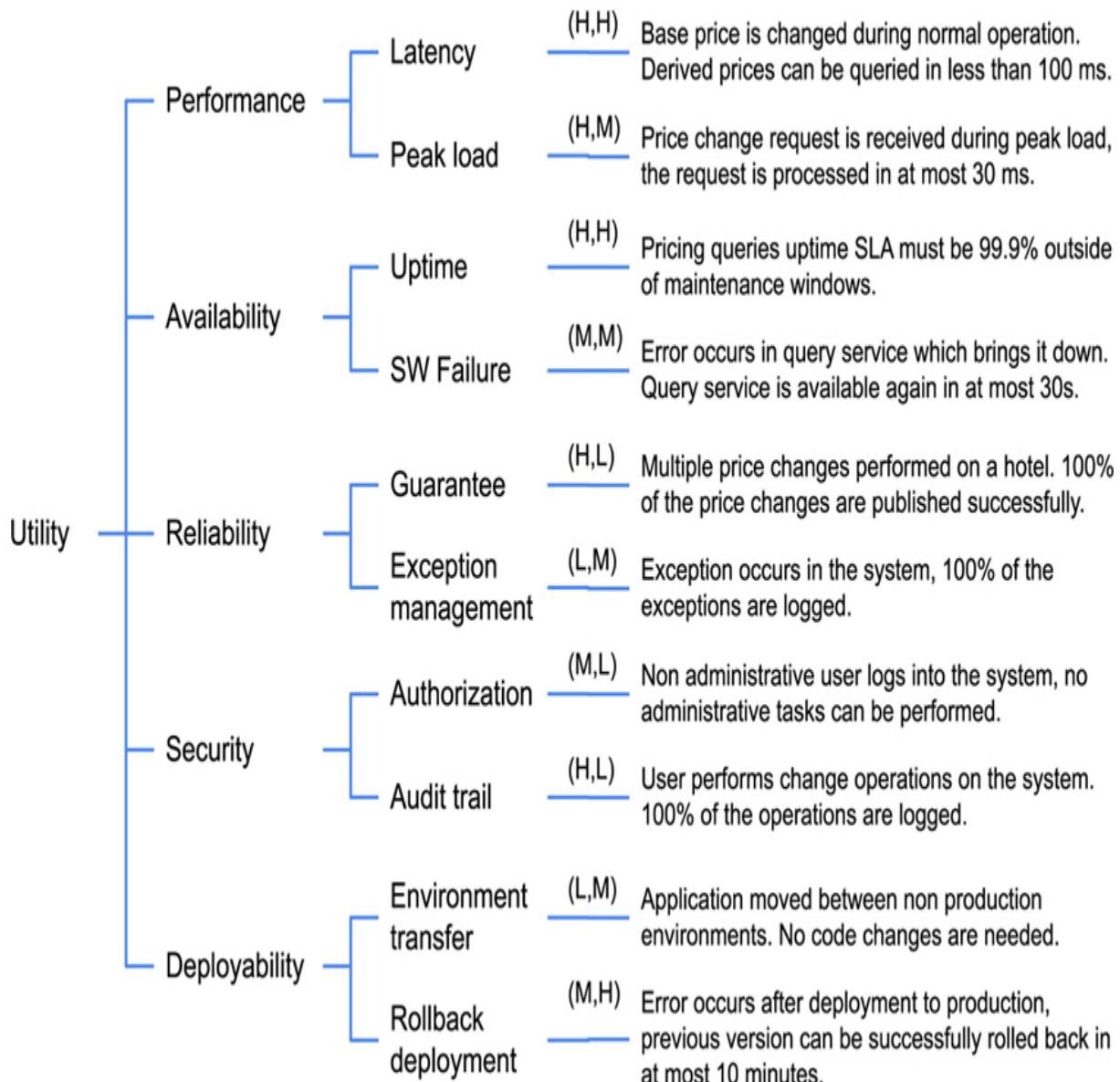


FIGURE 2.3 A Utility Tree

It works as follows. First write the word “Utility” on a sheet of paper. Then write the various quality attributes that constitute utility for your system. For example, you might know, based on the business goals for the system, that the most important qualities for the system are that the system be fast, secure, and easy to modify. In turn, you would write these words underneath “Utility.” Next, because we don’t really know what any of those terms actually means, we describe the aspect of the quality attribute that we are most

concerned with. For example, while “performance” is vague, “latency of price changes” is a bit less vague. Likewise, while “deployability” is vague, “rollback deployment” is a bit less vague.

The leaves of the tree are expressed as scenarios, which provide concrete examples of the quality attribute considerations that you just enumerated. For example, for “latency of price changes,” you might create a scenario such as “Base price is changed during normal operation. Derived prices can be queried in less than 100 ms.” For “deployability,” you might create a scenario such as “Error occurs after deployment to production, previous version can be successfully rolled back in at most 10 minutes.”

Finally, the scenarios that you have created must be prioritized. We do this prioritization by using the technique of ranking across two dimensions, resulting in a priority matrix such as the following (where the numbers in the cells are from a set of system scenarios).

Business Importance/Tech nical Risk	L	M	H
L	5, 6, 17, 20, 22	1, 14	12, 19
M	9, 12, 16	8, 20	3, 13, 15
H	10, 18, 21	4, 7	2, 11

Our job, as architects, is to focus on the lower-right-hand portion of this table (H, H): those scenarios that are of high business importance and high risk. Once we have satisfactorily addressed those scenarios, we can move to the (M, H) or (H, M) ones, and then move up and to the left until all of the system's scenarios are addressed (or perhaps until we run out of time or budget, as is often the case).

It should be noted that the QAW and the Utility Tree are two different techniques that are aimed at the same goal—eliciting and prioritizing the most important quality attribute requirements, which will be some of your most critical architectural drivers. There is no reason, however, to choose between these techniques. Both are useful and valuable and, in our experience, they have complementary strengths: The QAW tends to focus more on the requirements of external stakeholders, whereas the Utility Tree tends to excel at eliciting the requirements of internal stakeholders. Making all of these stakeholders happy will go a long way toward ensuring the success of your architecture.

2.4.3 Primary Functionality

Functionality is the ability of the system to do the work for which it was intended. As opposed to quality attributes, the way the system is structured does not normally influence functionality. You can have all of the functionality of a given system coded in a single enormous module (a god class, for example), or you can have it neatly distributed across many smaller, highly cohesive modules. Externally the system will look and work the same way if you consider only functionality. What matters, though, is what happens when you want to make changes to such a system. In the former case, changes will be difficult and costly; in the latter case, they should be much easier and cheaper to perform. In terms of architectural design, allocation of functionality to

elements, rather than the functionality per se, is what matters. A good architecture is one in which the most common changes are localized in a single or a few elements (as a consequence of high cohesion), and hence easy to make.

When designing an architecture, you need to consider at least the primary functionality. Primary functionality is usually defined as functionality that is critical to achieve the business goals that motivate the development of the system. These are typically captured as user stories or use cases and we will use these terms interchangeably through the remainder of this book. Primary functionality is often focused on the “happy” paths, where everything works as planned and the system delivers some useful outcome. Other criteria for primary functional requirements might be that it implies a high level of technical difficulty or that it requires the interaction of many architectural elements. As a rule of thumb, approximately 10 percent of your user stories or use cases are likely to be primary.

There are two important reasons why you need to consider primary functionality when designing an architecture:

1. You need to think how functionality will be allocated to elements (usually modules) to promote modifiability or reusability, and also to plan work assignments.
2. Some quality attribute scenarios are directly connected to the primary functionality in the system. For example, in a movie streaming application, one of the primary use cases is, of course, to watch a movie. This use case is associated with a performance quality attribute scenario such as “Once the user presses play, the movie should begin streaming in no more than 5 seconds.” In this case, the quality attribute scenario is directly associated with the primary user story, so making

decisions to support this scenario also requires making decisions about how its associated functionality will be supported. This is not the case for all quality attributes however. For example, an availability scenario can involve recovery from a system failure, and this failure may occur when any of the system's functions are being executed.

Decisions regarding the allocation of functionality that are made during architectural design establish a precedent for how the rest of the functionality should be allocated to modules as development progresses. This is usually not the work of the architect; instead, this activity is typically performed as part of the element interaction design process described in [Section 2.2.2](#).

Finally, bad decisions that are made regarding the allocation of functionality result in the accumulation of technical debt. This debt can be paid through the use of refactoring, although this impacts the project's rate of progress, or velocity (see [Chapter 10](#)).

2.4.4 Architectural Concerns

Architectural concerns encompass additional aspects that need to be considered as part of architectural design but that are not expressed as traditional requirements. There are several different types of concerns:

- *General concerns.* These are “broad” issues that one deals with in creating the architecture, such as establishing an overall system structure, the allocation of functionality to modules, the allocation of modules to teams and the organization of the code base.
- *Specific concerns.* These are more detailed system-internal issues such as exception management,

dependency management, configuration, logging, authentication, authorization, caching, and so forth that are common across large numbers of applications. Some specific concerns are addressed in reference architectures (see Section 2.5.1), but others will be unique to your system. Specific concerns also result from previous design decisions. For example, you may need to address API versioning if you previously decided to use a REST API.

- *Internal requirements.* These requirements are usually not specified explicitly in traditional requirement documents, as customers usually seldom express them. Internal requirements may address aspects that facilitate development, deployment, operation, or maintenance of the system. They are sometimes called “derived requirements.” An example of this would be to use a particular data type to manage currencies, or to store all timestamps in UTC.
- *Issues.* These result from analysis activities, such as a design review (see [Section 11.6](#)), so they may not be present initially. For instance, an architectural evaluation may uncover a security risk that requires some changes to be performed in the current design.

Some of the decisions surrounding architectural concerns might be trivial or obvious. For example, your deployment structure might be a single processor for an embedded system, or a single cell phone for an app. Your reference architecture might be constrained by company policy. Your authentication and authorization policies might be dictated by your enterprise architecture and realized in a shared framework. In other cases, however, the decisions required to satisfy particular concerns may be less obvious—for example, decisions involving exception management or input validation or structuring the code base.

From their past experience, wise architects are usually aware of the concerns that are associated with a particular type of system and the need to make early design decisions to address them. Inexperienced architects are usually less aware of such concerns; because these concerns tend to be tacit rather than explicit, they may not consider them as part of the design process, which often results in problems later on.

Architectural concerns frequently result in the introduction of new quality attribute scenarios. The concern of “supporting logging,” for example, is too vague and needs to be made more specific. Like the quality attribute scenarios that are provided by the customer, these scenarios need to be prioritized. For these scenarios, however, the customer is the development team, operations, or other members of the organization. During design, the architect must consider both the quality attribute scenarios that are provided by the customer and those scenarios that are derived from architectural concerns.

One of the goals of our revision of the ADD method was to elevate the importance of architectural concerns as explicit inputs to the architecture design process, as will be highlighted in our examples and case studies in [Chapters 8](#), and [9](#).

2.4.5 Constraints

You need to catalog the constraints on development as part of the architectural design process. These constraints may take the form of mandated technologies, other systems with which your system needs to interoperate or integrate, laws and standards that must be complied with, the abilities and availability of your developers, deadlines that are non-

negotiable, backward compatibility with older versions of systems, and so on. An example of a technical constraint is the use of open source technologies, whereas a nontechnical constraint is that the system must obey the Sarbanes-Oxley Act or that it must be delivered by December 15.

A constraint is a decision over which you have little or no control as an architect. Your job is, as we mentioned in [Chapter 1](#), to *satisfice*: to design the best system that you can, despite the constraints you face. Sometimes you might be able to argue for loosening a constraint, but in most cases you have no choice but to design around the constraints.

2.5 Summary

In this chapter, we introduced the idea of design as a set of decisions to satisfy requirements and constraints. We also introduced the notion of “architectural” design and showed that it does not differ from design in general, other than that it addresses the satisfaction of *architectural drivers*: the purpose, primary functional requirements, quality attribute requirements, architectural concerns, and constraints. What makes a decision “architectural”? A decision is architectural if it has nonlocal consequences *and* those consequences matter to the achievement of an architectural driver.

We also discussed why architectural design is so important: because it is the embodiment of early, far-reaching, hard-to-change decisions. These decisions will help you meet your architectural drivers, will determine much of your project’s work-breakdown structure, and will affect the tools, skills, and technologies needed to realize the system. Thus, architectural design decisions should be scrutinized well, as

their consequences are profound. In addition, architecture is a key enabler of agility.

2.6 Further Reading

A more in-depth treatment of scenarios and architectural drivers can be found in L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 4th ed., Addison-Wesley, 2021. Also found in this book is an extensive discussion of architectural tactics, which are useful in guiding an architecture to achieve quality attribute goals. Likewise, this book contains an extensive discussion of QAW and Utility Trees.

A discussion of technical debt, including design/architecture debt can be found in: N. Ernst, J. Delange, R. Kazman, *Technical Debt in Practice—How to Find It and Fix It*, MIT Press, 2021.

The Mission Thread Workshop is discussed in R. Kazman, M. Gagliardi, and W. Wood, “Scaling Up Software Architecture Analysis,” *Journal of Systems and Software*, 85, 1511–1519, 2012; and in M. Gagliardi, W. Wood, and T. Morrow, *Introduction to the Mission Thread Workshop*, Software Engineering Institute Technical Report CMU/SEI-2013-TR-003, 2013.

An overview of discovery prototyping, JRP, JAD, and accelerated systems analysis can be found in any competent book on systems analysis and design, such as J. Whitten and L. Bentley, *Systems Analysis and Design Methods*, 7th ed., McGraw-Hill, 2007.

An extensive collection of architectural design patterns for the construction of distributed systems can be found in F. Buschmann, K. Henney, and D. Schmidt, *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*, Wiley, 2007. Other

books in the POSA (Patterns Of Software Architecture) series provide additional pattern catalogs. Many other pattern catalogs specializing in particular application domains and technologies exist. A few examples are listed here:

- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- E. Fernandez-Buglioni. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. Wiley, 2013.
- J. Gilbert, *Software Architecture Patterns for Serverless Systems: Architecting for innovation with events, autonomous services, and micro frontends*, Packt Publishing, 2021.
- H. Percival, B. Gregory, *Architecture Patterns with Python: Enabling Test-Driven Development, Domain-Driven Design, and Event-Driven Microservices*, O'Reilly, 2020.

The “bible” for software architecture documentation is P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*, 2nd ed., Addison-Wesley, 2011.

A chapter devoted to software architecture is now included in Version 4.0 of the SWEBOK: Hironori Washizaki, eds., *Guide to the Software Engineering Body of Knowledge (SWEBOK)*, Version 4.0, IEEE Computer Society, 2024; www.swebok.org.

2.7 Discussion questions

1. Can you find an example of an application which, at the time of its development, was considered to have a novel design? Why was its design novel?
2. What is the risk of an architect doing all of the design of the system (including element interaction design and element internals design)? What happens if all of this design is performed initially, before any development begins?
3. Consider an application that you are familiar with. This might be one that you have worked on, or one you have used, such as a social network, a video streaming app, a productivity tool, a game, etc. What are drivers for this application?
4. Which functional requirement would be primary in the application that you selected? Why?
5. Which quality attribute would have the highest priority? Why?
6. Can you identify a constraint for this system?
7. Can you identify an obvious tradeoff that was made in the design of the application that you selected?

3. Making Design Decisions

Design is the process of making decisions. Which decisions? Ideally, the ones that will lead to project success! But decision-making is hard. It is fraught with uncertainty. We are often uncertain about our project requirements and project resources. We are uncertain about the evolutionary trajectories of the technologies we build upon. We are uncertain about our planning horizon. We may be uncertain about the costs of the choices we are making. We may even be uncertain about our own knowledge (although architects and others often fail in the other direction, being over-confident rather than under-confident—this is known as overconfidence bias).

As architects we are responsible for the decisions that we make. This is as it should be. So what practices are available to us to ensure that the decisions that we make are the best ones, given our abilities? That is the subject of this chapter. We begin by reviewing a set of decision-making techniques. Then we turn to the body of knowledge that we can employ, as architects, to make better, more confident decisions.

Why read this chapter?

You come to this chapter, we are certain, with some preconceived ideas about what design is and its role, but very few people have the depth and breadth of experience to help them make design decisions confidently. In this chapter we provide a sampling of the kinds of knowledge that experienced designers possess, and we will show how this knowledge forms the basis for making good design decisions with confidence. Having read this chapter you won't be an expert in design concepts--that is too much to expect of a single chapter--but you'll understand this essential tool that will be used throughout the book.

3.1 Making Design Decisions

We are seldom explicitly taught how to make decisions. It is somehow assumed that, given a proper education, we will make prudent choices. But this is not necessarily so. The field of design is littered with examples of “what were they thinking?!”. The architects who are really good decision-makers are often ones with decades of experience. They achieved their status by years of trial and error, by making mistakes, and by learning from their failures. This is clearly a sub-optimal form of training--expensive and inefficient and not very scalable. Can we do any better? We think that we can!

3.1.1 Decision-Making Principles

In what follows we present nine general decision-making principles. In following these principles, or strategies, you are not guaranteed to make a perfect decision. That is clearly too high a bar. But you can certainly increase the likelihood that the decision you are making is a sound, well-reasoned one which is why we recommend them. And while all of these principles are “common sense”, we see them

violated over and over again in practice. And the result of these (frequently avoidable) violations is projects with high technical debt, poor performance, poor scalability, stability issues, poor usability, and so on.

Principle 1: Use Facts

A fact is something that we believe to be true. Facts, and the evidence for them, are the foundations of logical reasoning. Incorrect or incomplete information may lead to invalid conclusions. An architect may hear praise for a new technology and base a decision on that, instead of testing it themselves. For example, a colleague of ours chose a NoSQL database in the Query side of a CQRS-based system (see [section 3.3.2](#)) he was designing because he felt it was a good fit; he had not done any prototyping or analysis, but made this decision based on hearsay, experience, and gut feelings. Was this the right choice? Time will tell... To check facts, he could have consciously considered: “What evidence supports the assumption that NoSQL will meet the system’s requirements now and in the future?”. When we cannot have all the facts, we make assumptions, as we will discuss next. We make a similar decision, as you will see, in [Chapter 8](#) (the case study where this decision was made). Was this the right decision?

Principle 2: Check Assumptions

In the absence of facts, we often make assumptions so that we can continue with the process of design. For instance, we may not know if a technology can perform adequately until we prototype the software. When we consciously make assumptions, we are saying that these assumptions will be good ones. Explicit assumptions can be checked. *Implicit assumptions*, on the other hand, are more devious. We may even be unaware we have made them. For example, if we build our application with node.js based on an existing

three-tier client-server architecture (which contains a monolithic database) without assessing the compatibility with the database's performance characteristics, then this implicit assumption would not be checked, and that might create a performance risk for the system.

Principle 3: Explore Contexts

Contexts are *conditions* that influence software decisions. There are many contextual factors such as development resources, financial pressures, legal obligations, industry norms, user expectations, and past decisions. For example, we might want to implement a scalable and highly reliable database system but our budget is limited. The budget is not a system requirement but it affects our decision on database license procurement. Some contexts will end up being constraints on design, such as team experience or expectations. Design contexts shape our decisions implicitly and these factors are not necessarily technology related. Exploring contextual factors can broaden our design considerations. To check that we have considered contexts we may ask: "*What are the contexts that could influence X?*", "*Have I missed any contexts?*", and "*Does the team have the experience in implementing X?*".

Principle 4: Anticipate Risks

A risk is the possibility of an undesirable outcome. A documented risk contains an estimate of the *size of the loss* and the *probability of the loss*. There are many risks that an architect needs to estimate such as extreme spikes of demand and security attacks. Anticipating and quantifying risks is the process of exploring the unknowns, estimating the possibility of risks occurring and, if they occur, their impacts. Architects may reflect on risks by asking questions such as: "*What are the potential undesirable outcomes?*", or "*Is there a chance that X would not work?*".

Principle 5: Assign Priorities

Priorities quantify the *relative importance* of choices, choices such as which requirement to implement or which solution to use. If we can only afford to implement one of the two requirements, which one is more important? Prioritization is required when the things that we desire are competing for the same limited resource such as time, money, developer skills, CPU, memory, network bandwidth. Some of these priorities emanate from contextual factors. To sort out our priorities we can ask: “*Which requirement is more important?*”, “*What can we do without?*”, “*What should we use this resource for?*”.

Principle 6: Define Time Horizon

The time horizon defines the *time period* relevant to a decision, and its impacts. Risks, benefits, costs, needs and impacts can change over time and we want to anticipate how they change. For example, we might estimate that the system processing load will reach 85% capacity in 3 years, or we might choose to deploy on premises in the short term while keeping alive the option to deploy on a public cloud if it becomes more cost-effective.

Defining the time horizon allows architects to explicitly state and evaluate the pros and cons of actions (and non-actions) in terms of their short- and long-term impacts. Without explicit considerations of time horizon and reasoning with it, long-term considerations may be undermined, or short-term needs may be ignored. The architect may ask questions such as: “*What would be affected in the short- and long-term if I decide on X?*”, or “*What needs to be considered in different time-horizons for X?*”

Principle 7: Generate Multiple Solution Options

Some architects will go with the first solution they think of without considering further options. If the architect is experienced and the problem is well understood and low-risk, this may be ideal. But in more challenging contexts a single solution may be risky; the first solution is not necessarily the best solution, especially when an architect is inexperienced or facing an unfamiliar situation. This behavior may be due to anchoring bias—a refusal to let go of the first idea. Generating multiple solution options helps an architect broaden solution choices and stimulate creativity. We can broaden solution ideas by asking ourselves questions such as: “*Are there other solutions to this problem?*”, or “*Can I find a better solution than X?*”.

Principle 8: Design Around Constraints

Constraints are *limitations* that set the boundaries of our design options. Constraints may come from requirements, contexts, technologies as well as existing design choices. For instance, a CPU can only compute W instructions per second; the budget of the project is \$X; the number of concurrent users supported by a software license is Y; platform Z doesn’t support a certain protocol; developers have no experience with some technology, and so forth. In software development, we often find connected sets of constraints: if we choose component A, we must also use component B. When there are no apparent solutions architects must design around constraints and introduce novel solutions, relax constraints, or manipulate the contexts. An architect can check on constraints by asking: “*If I choose X, would it limit other design options?*”, “*Are there any constraints that could impede this design choice?*”.

Principle 9: Weigh Pros and Cons

Pros and cons represent the arguments *for and against* each of the choices in a selection. Weighing pros and cons is heavily bound up with examining trade-offs. The evaluation of the pros and cons allows architects to decide what to choose and what to avoid. A quantitative evaluation can be based on measurable elements such as costs, benefits, priorities, immediacy (i.e. time horizon) and risks. However, some pros and cons cannot be easily quantified. Consider the navigation menu design of a mobile app; how can one quantify the pros and cons of a hamburger menu versus a set of tabs? In this case, qualitative arguments such as ease of access, ease of learning, effort to implement can be marshaled. Weighing pros and cons offers architects the chance to think about relative benefits and drawbacks and to whom they affect. To check trade-offs, one can ask questions such as: “*Are there more benefits in solution X than Y?*”, or “*Have we done a thorough trade-off evaluation?*”.

Now that we have enumerated a set of principles, let us turn our attention to the design concepts that we use to structure the system and to support our most important quality attributes.

3.2 Design Concepts: The Building Blocks for Creating Structures

Design is not random, but rather is planned, intentional, rational, and directed. The process of design may seem daunting at first. When facing the “blank page” at the beginning of any design iteration, the space of possibilities might seem impossibly huge and complex; however, there is some help here. The software architecture community has created and evolved, over the course of decades, a body of

generally accepted design principles that can guide us to create high-quality designs with predictable outcomes.

For example, some well-documented design principles are oriented toward the achievement of specific quality attributes:

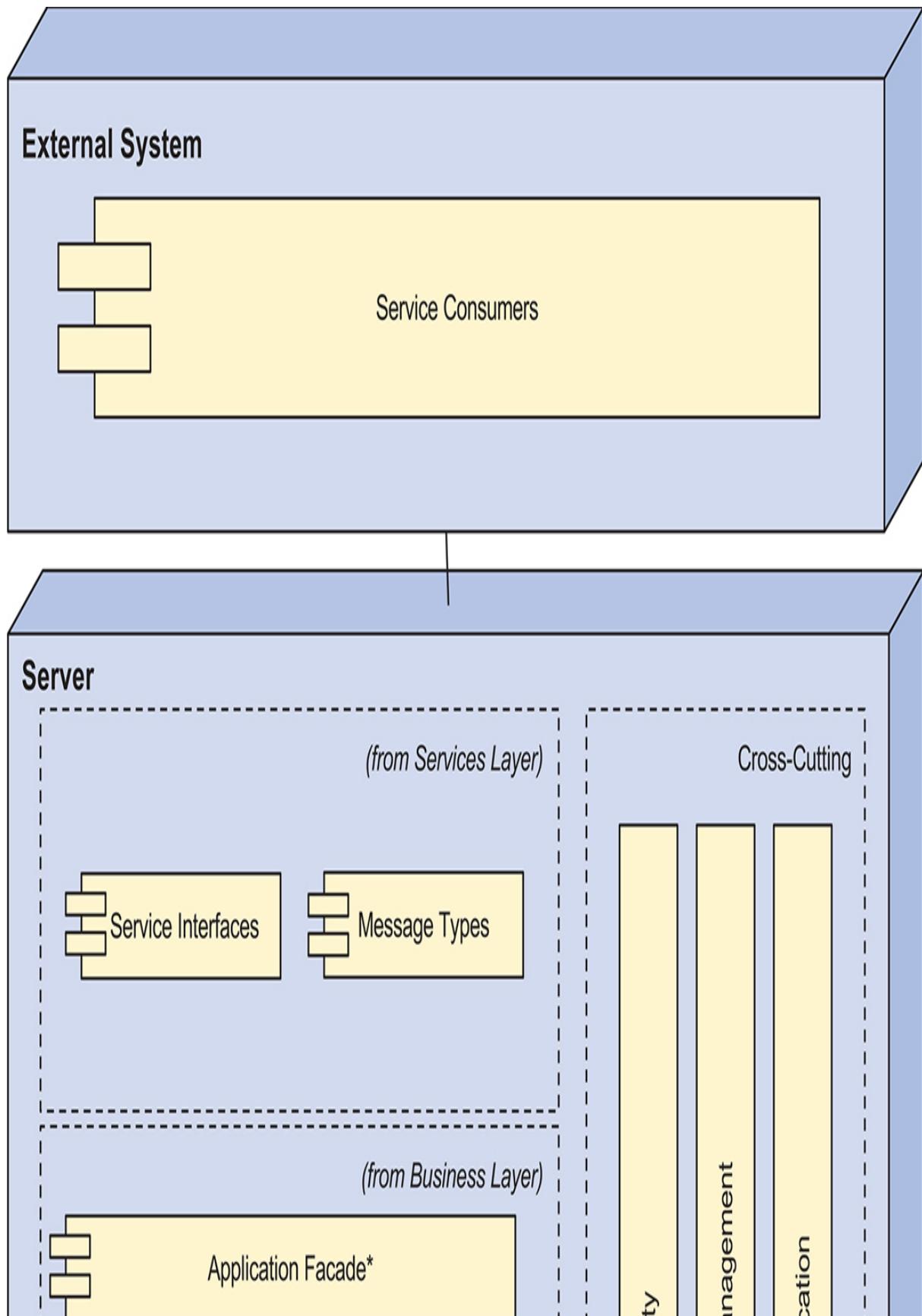
- To help achieve high modifiability, aim for good modularity, which means high cohesion and low coupling.
- To help achieve high availability, avoid having any single point of failure.
- To help achieve scalability, avoid having any hard-coded limits for critical resources.
- To help achieve security, limit the points of access to critical resources.
- To help achieve testability, externalize state.
- . . . and so forth.

In each case, these principles have been evolved over decades of dealing with those quality attributes in practice. But these principles are rather abstract. To make them more usable in practice, a set of *design concepts* have been cataloged. They are the building blocks of architectures. Different types of design concepts exist, and in the following subsections we discuss some of the most commonly used, including reference architectures, patterns, tactics, and externally developed components (such as frameworks). While the first three are conceptual in nature, the last one is concrete.

3.2.1 Reference architectures

Reference architectures are blueprints that provide an overall logical structure for particular classes of applications. A reference architecture is a reference model mapped onto one or more architectural patterns. It has been proven in business and technical contexts, and typically comes with a set of supporting artifacts that eases its use.

An example of a reference architecture for the development of service applications (such as microservices) is shown in [Figure 3.1](#). This reference architecture establishes the main layers for this type of application—service, business, and data—as well as the types of elements that occur within the layers and the responsibilities of these elements, such as service interfaces, business components, data access components, service agents, and so on. Also, this reference architecture introduces cross-cutting concerns, such as security and communication, that need to be addressed. As this example shows, when you select a reference architecture for your application, you also adopt a set of concerns that you need to address during design. You may not have an explicit requirement related to communications or security, but the fact that these elements are part of the reference architecture guide you to make design decisions about them.



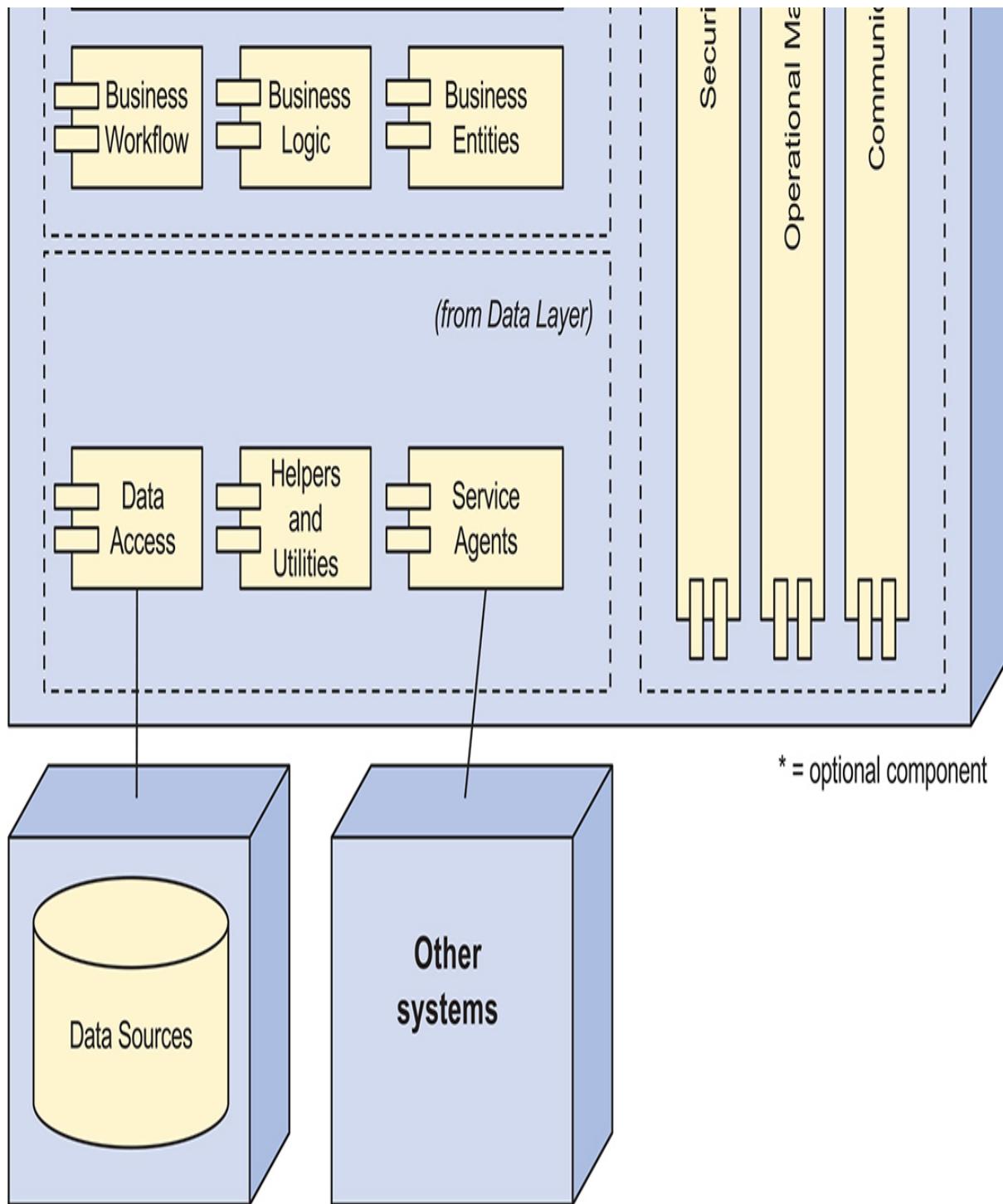


FIGURE 3.1 Service Application Reference Architecture

Reference architectures may be confused with architectural styles, but these two concepts are different. Architectural styles (such as “Pipe and Filter” and “Client Server”) are

patterns that define types of components and connectors in a specified topology that are useful for structuring an application either logically or physically. Reference architectures, in contrast, provide a more detailed structure for entire applications, and they may embody multiple patterns. Reference architectures are preferred by practitioners—which is also why we favor them in our list of design concepts.

There are many reference architectures, but there is no single authoritative catalog of them. In the context of cloud development, providers have created a number of useful catalogs that collect reference architectures focused on cloud resources. There are, in addition, many catalogs of patterns.

3.2.2 Patterns

Design/architectural patterns are conceptual solutions to recurring design problems that exist in a defined context. While design patterns originally focused on decisions at the object scale, including instantiation, structuring, and behavior, today there are catalogs with patterns that address decisions at varying levels of granularity. In addition, there are specific patterns to address quality attributes such as security or availability or performance or integrability.

While some people argue for the differentiation between what they consider to be architectural patterns and the more fine-grained design patterns, we believe there is no principled difference that can be solely attributed to scale. We consider a pattern to be architectural when its use directly and substantially influences the satisfaction of some of a system's architectural drivers.

Another important type of pattern that has significant implications for the architectural design process is deployment patterns. These patterns package key decisions that shape the system's infrastructure and have profound implications for quality attributes such as availability, performance, modifiability or usability. Such decisions are often bound early in the product life cycle. We will discuss deployment patterns in [chapter 6](#).

We provide examples of patterns for a number of quality attributes in [sections 3.3](#) through [3.8](#). But these are just examples, and many more patterns exist for each quality attribute, and for other qualities.

3.2.3 Tactics

Architects can also employ fundamental design techniques to help achieve a response goal for a particular quality attribute. We call these architectural design primitives *tactics*. Tactics, like patterns, are techniques that architects have been using for years. We do not invent tactics, but simply capture what architects actually have done in practice, over the decades, to manage quality attribute response goals.

Tactics are design decisions that influence the control of a quality attribute response. For example, if you want to design a system to have good usability, you would need to make a set of design decisions that would support this quality attribute, as represented in [Figure 3.2](#).

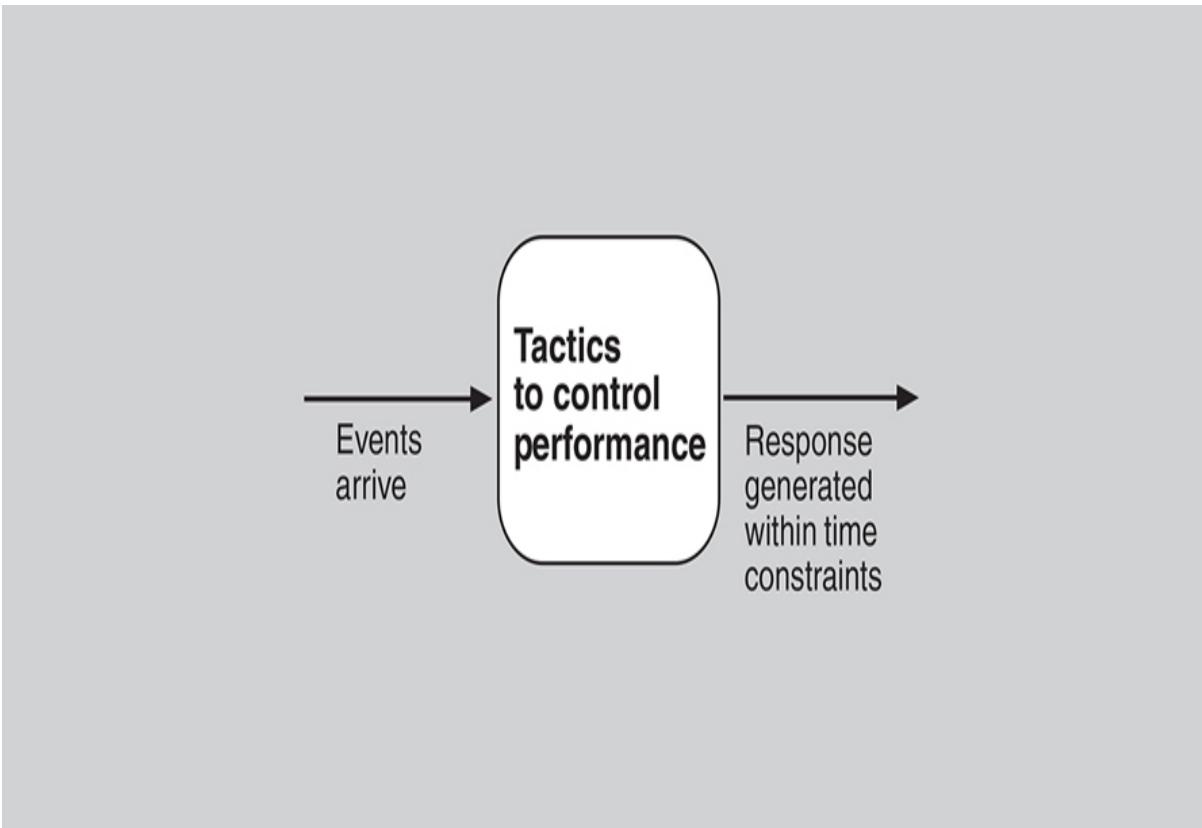


FIGURE 3.2 *Usability Tactics mediate user interactions and responses*

Tactics are simpler than patterns. They focus on the control of a single quality attribute response (although they may, of course, trade off this response with other quality attribute goals). Patterns, in contrast, typically focus on resolving and balancing multiple forces—that is, multiple quality attribute goals. By way of analogy, we can say that a tactic is an atom, whereas a pattern is a molecule.

Tactics provide a top-down way of thinking about design. A tactics categorization begins with a set of design objectives related to the achievement of a quality attribute, and presents the architect with a set of options from which to choose.

For example, in [Figure 3.3](#), the design objectives for usability are “Support user initiative” and “Support system

initiative". An architect who wants to create a system with "good" usability needs to choose one or more of these options. That is, the architect needs to decide if they want to support a user in their tasks by allowing for features such as undo, cancel (for long-running or hung operations), aggregation (of similar UI objects so that a user-issued command applies to all of them) or pause/resume (again, typically used for long-running operations). And the architect may also want to support system initiatives, such as maintaining a model of the task (so that, for example, context-appropriate help and guidance can be given), maintaining a model of the user (so that user-appropriate feedback is offered; for example, a novice user might want far more guidance, whereas an expert user might want shortcuts for common operations), or maintaining a system model (for example, being able to accurately estimate time remaining for long-running operations).

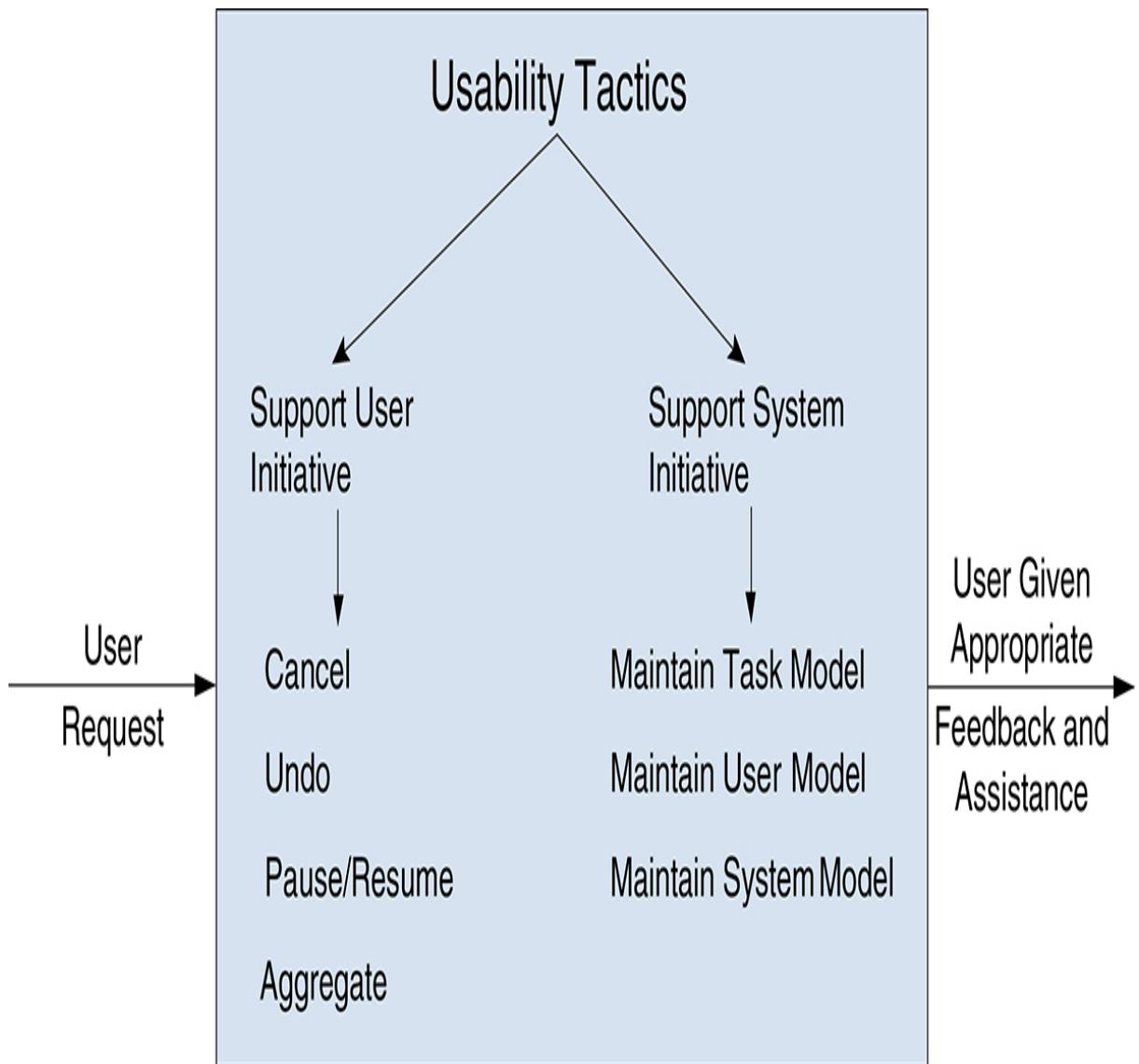


FIGURE 3.3 Usability Tactics Categorization

Each of these tactics is an option for the architect. They may be instantiated via coding, via patterns, or via external components such as frameworks. But they are architectural. To offer undo capabilities the system must maintain a set of transactions, representing changes to the system state, and be able to roll back any of those transactions to undo, that is to return the system to its prior state. To be able to offer “cancel” the system must be able to revert its state to whatever it was before the operation was initiated. These sorts of capabilities require architectural thinking, and

tactics provide a starting point for this decision-making process. As we will see in [Chapter 4](#), the choice, combination, and tailoring of tactics and patterns are some of the key steps of the ADD process.

There are established tactics categorizations for the quality attributes of availability, deployability, energy efficiency, interoperability, modifiability, performance, safety, security, testability, and usability. Collectively these categorizations cover the vast majority of the design decisions that an architect needs to make in practice.

3.2.4 Externally developed components

Patterns and tactics are, however, abstract in nature. When you are designing a software architecture, you need to make these design concepts concrete and closer to the actual implementation. There are two ways to achieve this: You can code the elements obtained from tactics and patterns or you can associate technologies with one or more of these elements in the architecture. This “buy versus build” choice is one of the most important decisions you will make as an architect.

We consider technologies to be *externally developed components*, because they are not created as part of the development project. Several types of externally developed components exist:

- *Technology families*. A technology family represents a class of technologies with common functional purposes. It can serve as a placeholder until a specific product or framework is selected. An example is a relational database management system (RDBMS) or an object-oriented to relational mapper (ORM).

- *Products.* A product (or software package) refers to a self-contained piece of software that can be integrated into the system that is being designed and that requires only minor configuration or coding. An example is a relational database management system, such as Oracle or PostgreSQL, which belong to the RDBMS technology family. APIs (Application Programming Interfaces) from external systems are one common type of product; we discuss them in [Chapter 5](#). We discuss another type of product: cloud capabilities made available by cloud provider platforms in [Chapter 7](#).
- *Application frameworks.* An application framework (or just framework) is a reusable software element, constructed out of patterns and tactics, that provides generic functionality addressing recurring domain and quality attribute concerns across a broad range of applications. Frameworks, when carefully chosen and properly implemented, increase the productivity of programmers. They do so by enabling programmers to focus on business logic and end-user value, rather than underlying technologies and their implementations. As opposed to products, framework functions are generally invoked from the application code or are “injected” using some type of dependency injection approach. Frameworks usually require extensive configuration, typically through XML files or other approaches such as annotations in Java. A framework example is Hibernate, which is used to perform object-oriented to relational mapping in Java. Several types of frameworks are available: Full-stack frameworks, such as Spring, are usually associated with reference architectures and address general concerns across the different elements of the reference architecture, while non-full-stack frameworks, such as JPA (Java Persistence API), address specific functional or quality attribute concerns.

- *Platforms*. A platform provides a complete infrastructure upon which to build and execute applications. Examples of current cloud platforms include Amazon Elastic Beanstalk, Google App Engine, Azure App Service which are all Platform as a Service (PaaS) offerings.

The selection of externally developed components, which is a key aspect of the design process, can be a challenging task because of their extensive number. Here are a few criteria you should consider when selecting externally developed components:

- *Problem that it addresses*. Is it something specific, such as a framework for object-oriented to relational mapping or something more generic, such as a complete platform?
- *Cost*. What is the cost of the license and, if it is free, what is the cost of support and education?
- *Type of license*. Does it have a license that is compatible with the project goals?
- *Vendor lock-in*. Will the inclusion of the technology result in a dependency on the organization that produces it?
- *Learning curve*. Is this a technology that is difficult to learn? Is it difficult to find developers that have some expertise with it?
- *Community support*. Is this a technology that is well supported by a robust community or by its vendor?

In the following sections, we focus our attention on two categories of design concepts: tactics and patterns. We focus on these, and intentionally omit externally developed components. The choice of externally developed components is clearly an important design decision;

however, the rapid evolution of technologies makes them prone to become quickly outdated and this is why we do not discuss them extensively here.

3.3 Design Concepts to Support Performance

Performance is about time and the software system's ability to meet timing requirements. Generally speaking, faster is better!

When events occur—interrupts, messages, requests from users or other systems, or clock events—the system must respond to them in time. Characterizing the events that can occur (and when they can occur) and the system's response to those events is the starting point for discussing performance. All systems have performance requirements, even if they are not expressed.

Performance is often linked to scalability—increasing your system's capacity, while still performing well. Often, performance is considered after you have constructed something and found its performance to be inadequate. You can fix this by architecting your system consciously with performance in mind.

3.3.1 Performance Tactics

The performance tactics categorization is shown in [Figure 3.4](#). This set of tactics, as with all tactics, helps an architect reason about the quality attribute. There are two major categories of performance tactics: *Control Resource Demand* and *Manage Resources*.

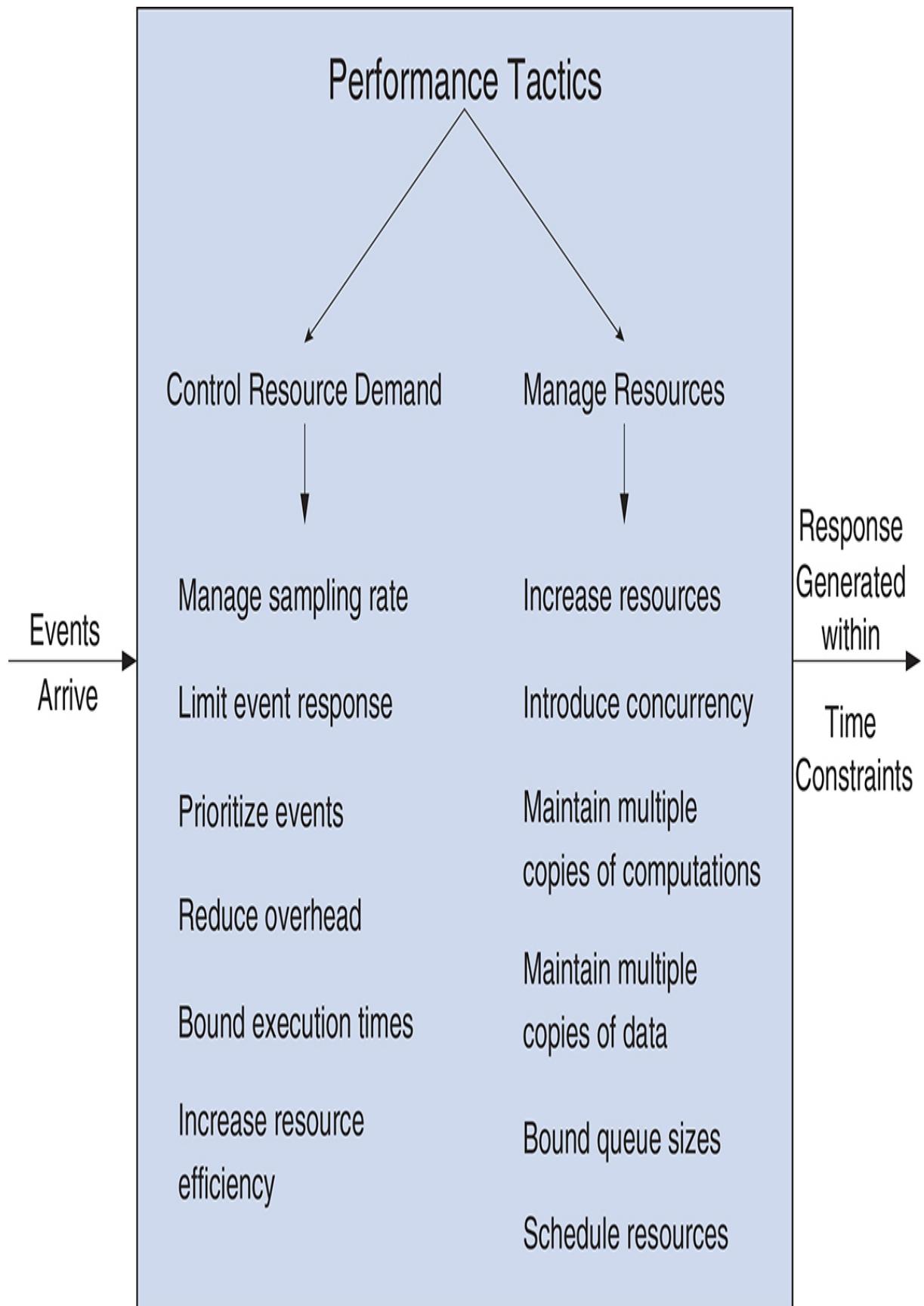


FIGURE 3.4 Performance Tactics Categorization

Within the *Control Resource Demand* category, the tactics are:

- *Manage work requests.* One way to reduce work is to reduce the number of requests coming into the system. Ways to do that include managing work requests (i.e. limiting the number of requests the system will accept in a given time period) and managing the sampling rate (for example, switching to a lower frame rate for streaming video).
- *Limit event response.* When events arrive too rapidly to be processed, they must be queued until they can be processed, or they are discarded. You may choose to process events only up to a set maximum rate, thereby ensuring predictable processing for other events.
- *Prioritize events.* If not all events are equally important, you can impose a priority scheme that ranks events according to how urgently you want to service them.
- *Reduce computational overhead.* For events that make it into the system, you can reduce the amount of work in handling each event by: reducing indirection (making direct calls rather than going through an intermediary, for example), by co-locating communicating resources, and by periodic cleaning (such as garbage collection).
- *Bound execution times.* You can place a limit on how much execution time is used to respond to an event.
- *Increase efficiency of resource usage.* Improving the efficiency of algorithms and data structures (adding an index to database, for example) in critical areas can decrease latency, improve throughput and resource consumption.

Within the *Manage Resources* category, the tactics are:

- *Increase resources.* Faster processors, additional processors, additional memory, and faster networks all have the potential to improve performance.
- *Introduce concurrency.* If requests can be processed in parallel, blocked time can be reduced.
- *Maintain multiple copies of computations.* This tactic reduces the contention that would occur if all requests were allocated to a single instance.
- *Maintain multiple copies of data.* Two common examples of maintaining multiple copies of data are data replication and caching.
- *Bound queue sizes.* This tactic controls the maximum number of queued arrivals and consequently the resources used to process the arrivals.
- *Schedule resources.* Whenever contention for a resource occurs, the resource should be scheduled. Your concern is to understand the characteristics of each resource's use and choose an appropriate scheduling strategy.

These tactics cover the spectrum of architectural concerns with respect to performance. Now we turn our attention to more complex design structures, patterns.

3.3.2 Performance Patterns

In what follows we provide a small selection of architectural patterns that address performance concerns. We make no attempt here to provide a comprehensive catalog; that is not the purpose of this book. And performance patterns are described abundantly in other resources. We merely provide a few patterns here to stimulate thinking and to provide

examples of the kinds of resources that are available to support the architect in reasoning about design for performance.

3.3.2.1 Load Balancer Pattern

A load balancer is an intermediary that handles messages from clients and determines which instance of a service should respond. The load balancer serves as a single point of contact for incoming messages and farms out requests to a pool of (typically stateless) providers.

By sharing the load among a pool of providers, latency can be kept lower and more predictable for clients. It is simple to add more resources to the resource pool, and no client needs to be aware of this (an instance of the Increase resources, Maintain multiple copies of computations, and Introduce concurrency tactics). And any failure of a server is invisible to clients, assuming there are still some remaining processing resources; note that this is an availability benefit but, as we mentioned earlier, patterns often address multiple quality goals whereas tactics address just a single goal.

The load balancing algorithm must be very fast; otherwise, it may itself contribute to performance problems. The load balancer is a potential bottleneck or single point of failure, so it is often replicated (and even load balanced).

3.3.2.2 Throttling Pattern

The throttling pattern packages the Manage work requests tactic. It is used to limit access to an important service. In this pattern, there is an intermediary—a throttler—that monitors the service and determines whether an incoming request can be serviced. By throttling incoming requests, you can gracefully handle variations in demand. In doing so,

services never become overloaded; they can be kept in a performance “sweet spot” where they handle requests efficiently.

But, once again, there are tradeoffs to consider: The throttling logic must be very fast; otherwise, it may itself contribute to performance problems. If client demand regularly exceeds capacity, buffers will need to be large, or you may lose requests. And this pattern can be difficult to add to an existing system where clients and servers are tightly coupled.

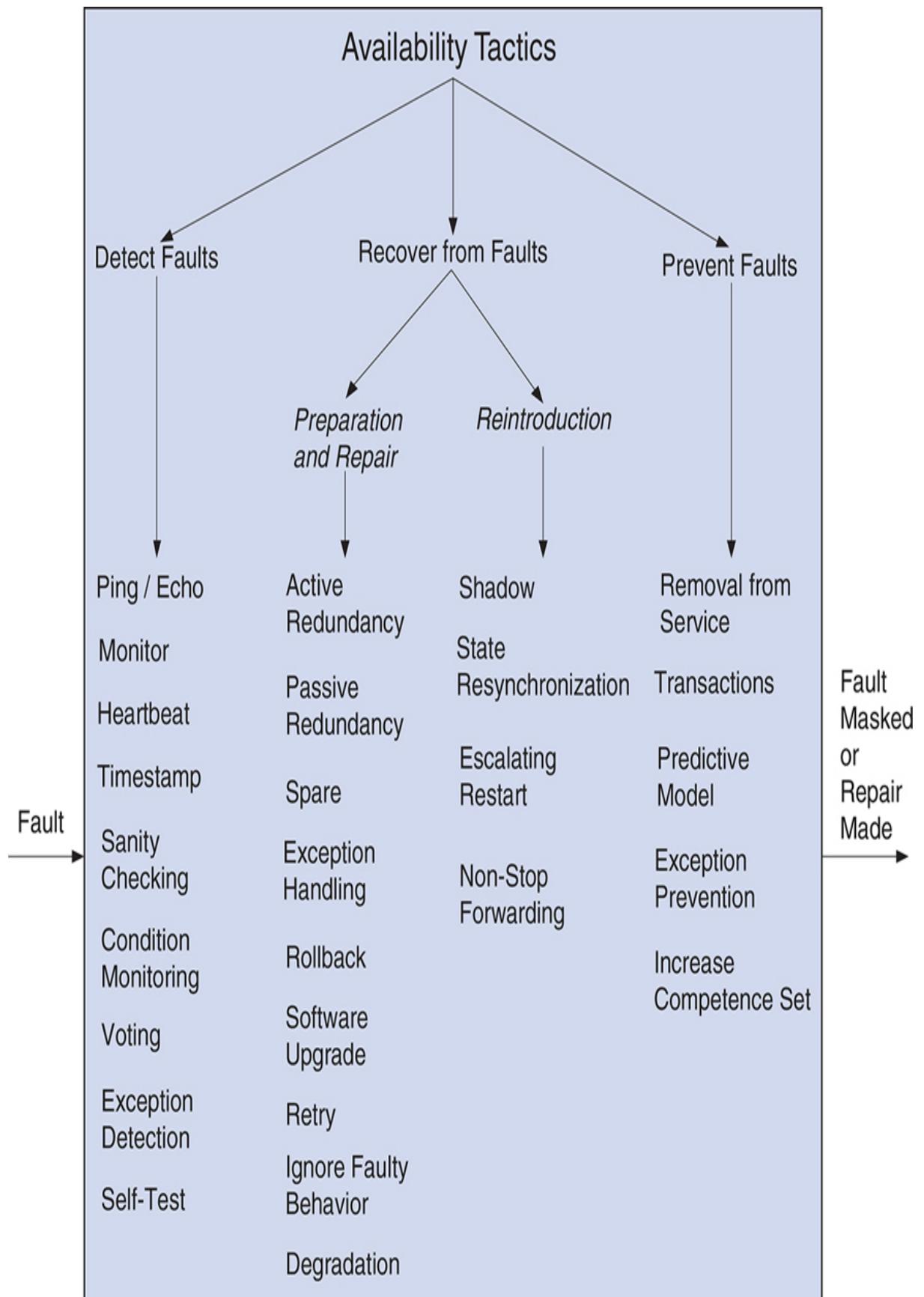
3.4 Design Concepts to Support Availability

Availability is a system property. A highly available system is compliant with its specifications—always ready to carry out its tasks when you need it to be. A *failure* in a system occurs when the system no longer delivers a service consistent with its specification. A *fault* has the potential to cause a failure. Availability encompasses the ability of a system to mask or repair faults so they do not become failures. Availability builds on the quality attribute of reliability by adding the notion of recovery (repair). The goal is to minimize service outage time by mitigating faults.

3.4.1 Availability Tactics

Availability tactics enable a system to endure faults. The tactics keep faults from becoming failures (or bound the effects of the fault and make repairs). A failure’s cause is a fault. A fault can be internal or external to the system. Faults can be: prevented, tolerated, removed, or forecast. Through these actions, a system can become “resilient” to faults.

[Figure 3.5](#) shows the availability tactics categorization. There are three major categories of availability tactics: *Detect Faults*, *Recover from Faults*, and *Prevent Faults*. It is difficult to envision a system that achieves high availability if it does not address all of these categories.



Reconfiguration

FIGURE 3.5 Availability Tactics Categorization

When designing for availability we are concerned with: how faults are detected, how frequently they occur, what happens when they occur, how long a system may be out of operation, how faults or failures can be prevented, and what notifications are required when a failure occurs. Each of these concerns can be addressed architecturally, through the appropriate selection of availability tactics.

Within the Detect Faults category, the tactics are:

- *Monitor*: a component used to monitor the state of health of other parts of the system. A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack.
- *Ping/echo*: asynchronous request/response message pair exchanged between nodes, used to determine reachability and the round-trip delay through the associated network path.
- *Heartbeat*: a periodic message exchange between a system monitor and a process being monitored.
- *Timestamp*: used to detect incorrect sequences of events, primarily in distributed message-passing systems.
- *Condition Monitoring*: checking conditions in a process or device, or validating assumptions made during the design.
- *Sanity Checking*: checks the validity or reasonableness of a component's operations or outputs; typically based

on a knowledge of the internal design, the state of the system, or the nature of the information under scrutiny.

- *Voting*: to check that replicated components are producing the same results. Comes in various flavors: replication, functional redundancy, analytic redundancy.
- *Exception Detection*: detection of a system condition that alters the normal flow of execution, e.g. system exception, parameter fence, parameter typing, timeout.
- *Self-test*: procedure for a component to test itself for correct operation.

Within the Recover from faults category there are two sub-categories, “Preparation and Repair” and “Reintroduction”. Let us first look at the tactics for “Preparation and Repair”. These are:

- *Redundant spare*: a configuration in which one or more duplicate components can step in and take over the work if the primary component fails. (This tactic is at the heart of the hot spare, warm spare, and cold spare patterns, which differ primarily in how up-to-date the backup component is at the time of its takeover, as we will discuss in [section 3.4.2](#).).
- *Rollback*: revert to a previous known good state, referred to as the “rollback line”.
- *Exception Handling*: dealing with the exception by reporting it or handling it, potentially masking the fault by correcting the cause of the exception and retrying.
- *Software Upgrade*: in-service upgrades to executable code images in a non-service-affecting manner.
- *Retry*: where a failure is transient, retrying the operation may lead to success.

- *Ignore Faulty Behavior*: ignoring messages sent from a source when it is determined that those messages are spurious.
- *Graceful Degradation*: maintains the most critical system functions in the presence of component failures, dropping less critical functions.
- *Reconfiguration*: reassigning responsibilities to the resources left functioning, while maintaining as much functionality as possible.

And within the “Reintroduction” sub-category, the tactics are:

- *Shadow*: running a previously failed/upgraded component in “shadow mode” prior to reverting it to an active role.
- *State Resynchronization*: partner to the redundant spare tactic, where state is sent from active to standby components.
- *Escalating Restart*: recover from faults by varying the granularity of the component(s) restarted.
- *Non-stop Forwarding*: functionality is split into supervisory and data. If a supervisor fails, a router continues forwarding packets along known routes while protocol information is recovered.

The final category of Availability tactics is Prevent Faults. The tactics within this category are:

- *Removal From Service*: temporarily placing a system component in an out-of-service state for the purpose of mitigating potential failures.
- *Transactions*: bundling state updates so that messages exchanged between components are atomic, consistent, isolated, and durable.

- *Predictive Model*: monitoring the state of a process to ensure that the system is operating properly; taking corrective action when conditions are predictive of likely future faults.
- *Exception Prevention*: preventing system exceptions from occurring by masking a fault, or preventing it via mechanisms such as smart pointers, abstract data types, and wrappers.
- *Increase Competence Set*: designing a component to handle more cases—faults—as part of its normal operation.

3.4.2 Availability Patterns

There are a number of redundancy patterns that are commonly used to achieve high availability. Let us look at them together, as this helps us understand their strengths and weaknesses, and the scope of the design space around this issue. These patterns deploy a group of active components and a group of redundant spare components. In general, the greater the level of redundancy, the higher the availability and the higher the cost and complexity.

The benefit of using a redundant spare is a system that continues to function correctly with only a brief delay after a failure. The alternative is a system that stops functioning correctly (or altogether) until the failed component is repaired.

The tradeoff with any of these patterns is the additional cost and complexity incurred in providing a spare. The tradeoff among the three alternatives is the time to recover from a failure versus the runtime cost incurred to keep a spare up-to-date. A hot spare carries the highest cost but leads to the fastest recovery time, for example.

3.4.2.1 Hot spare (Active redundancy)

This refers to a configuration in which all of the components belong to the active group and receive and process identical inputs in parallel, allowing the redundant spares to maintain a synchronous state with the active components. Because the redundant spare possesses an identical state to the active component, it can take over from a failed component in a matter of milliseconds.

3.4.2.2 Warm spare (Passive redundancy)

In this variant only the members of the active group process input. One of their duties is to provide the redundant spares with periodic state updates. Because this state is loosely coupled with the active components, the redundant components are referred to as warm spares.

Passive redundancy achieves a balance between the more highly available but more compute-intensive (and expensive) hot spare pattern and the less available but less complex (and cheaper) cold spare pattern.

3.4.2.3 Cold spare (Spare)

Cold sparing refers to a configuration where spares remain out of service until a failure, at which point a power-on-reset is initiated on the spare prior to it being placed in service. Due to its poor recovery performance, and hence its high MTTR (Mean Time to Recovery), this pattern is poorly suited to systems with high-availability requirements.

3.4.2.4 TMR (Tri-Modular Redundancy)

This widely used implementation of the Voting tactic, combined with the hot spare pattern, employs three identical components (or, more generally, N identical

components in N-modular redundancy). An example is shown in [Figure 3.6](#). Here each component receives the same inputs and forwards its output to the voter. If the voter detects any inconsistency, it reports a fault. For this reason, the value of N is usually set to an odd number, to avoid ties in the voting. The voter must also decide which output to use. Typical choices are letting the majority rule or choosing a computed average of the outputs.

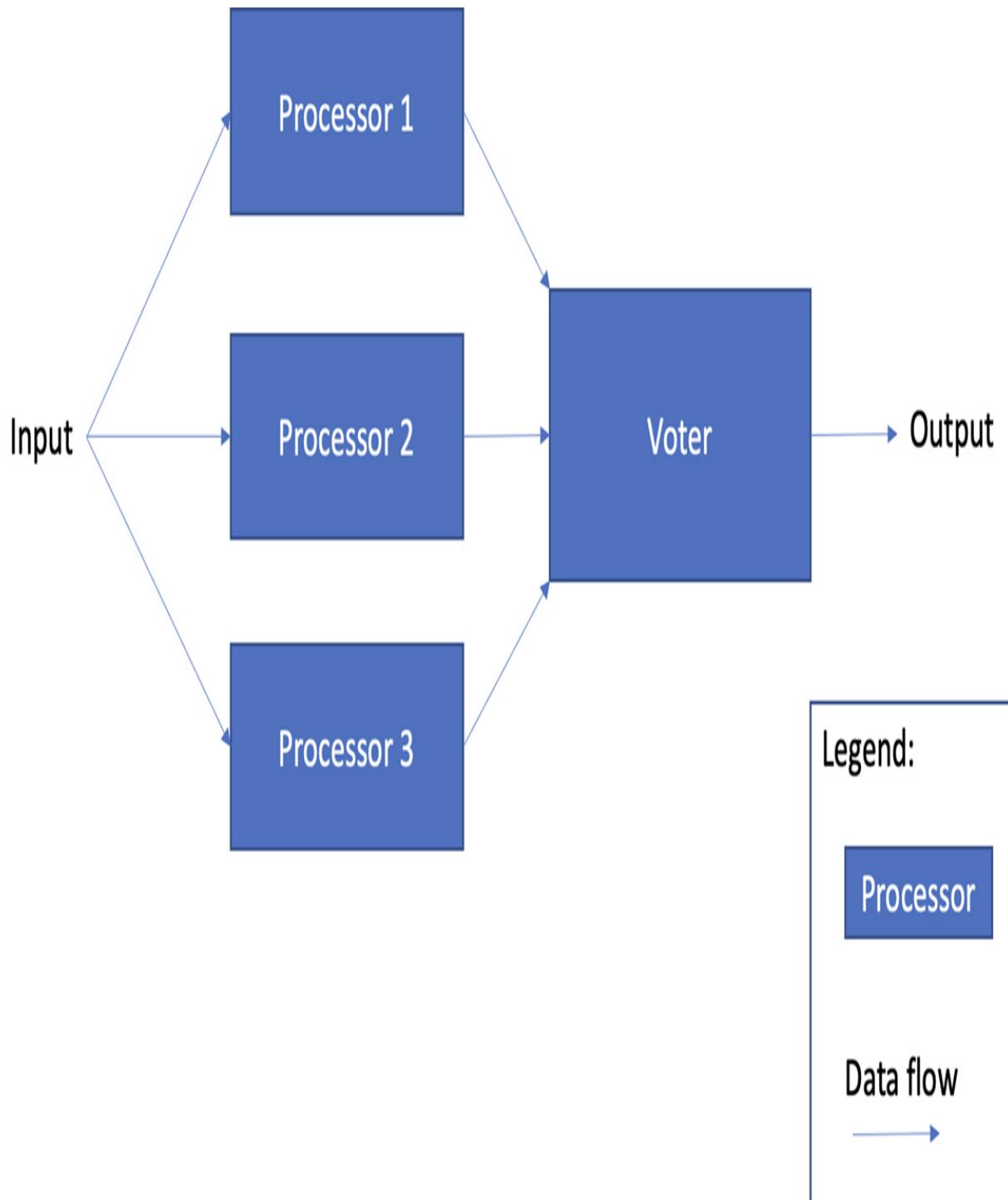


FIGURE 3.6 *Tri-Modular Redundancy pattern*

The TMR pattern is simple to understand and implement. It is independent of what might be causing disparate results

and is only concerned with making a choice so that the system can continue to function.

There is, as with the other redundancy patterns, a tradeoff between increasing the level of replication, which raises the cost, and the resulting availability. The statistical likelihood of two or more components failing is small, and three components is a common sweet spot between availability and cost.

3.4.2.5 Circuit Breaker

This next pattern is sometimes seen as a performance pattern. But it addresses both performance and availability concerns. This shows that the label on a pattern is less important than the response measures that it helps you to achieve. The circuit breaker pattern does not use redundancy, but it is important for achieving high availability in networked contexts, such as microservice architectures. A common availability tactic in networked systems is retry. For example, in the event of a timeout or fault when invoking a service, an invoker may try again (and again and again). A circuit breaker keeps the invoker from retrying infinitely many times, waiting for a response that may never come. This breaks the endless retry cycle when the circuit breaker deems that the system is dealing with a fault. Until the circuit is “reset,” subsequent invocations will return immediately without requesting the service.

One important benefit of this pattern is that it removes from individual components the policy of how many retries to allow before declaring a failure. The circuit breaker, in conjunction with software that listens to it and begins recovery, prevents cascading failures.

But care must be taken in choosing timeout (or retry) values. If the timeout is too long, then unnecessary latency

is added. If the timeout is too short, then the circuit breaker will trip when it does not need to, which can lower the availability and performance of services.

3.5 Design Concepts to Support Modifiability

Modifiability is about change. As architects we need to consider and plan for the cost and risk of making anticipated changes. To be able to plan for modifiability, an architect has to consider three big questions: 1) What can change? 2) What is the likelihood of the change? and 3) When is the change made and who makes it?

Change is so prevalent in the life of software systems that special names have been given to specific flavors of modifiability. Some of the common ones are scalability, variability, portability, and location independence, but there are many others.

3.5.1 Modifiability Tactics

Architects need to worry about modifiability to make the system easy to understand, debug, and extend. Tactics can help address these concerns.

The modifiability tactics categorization is shown in [Figure 3.7](#). There are three major categories of modifiability tactics: *Increase Cohesion*, *Reduce Coupling*, and *Defer Binding*. Let us examine these in turn.

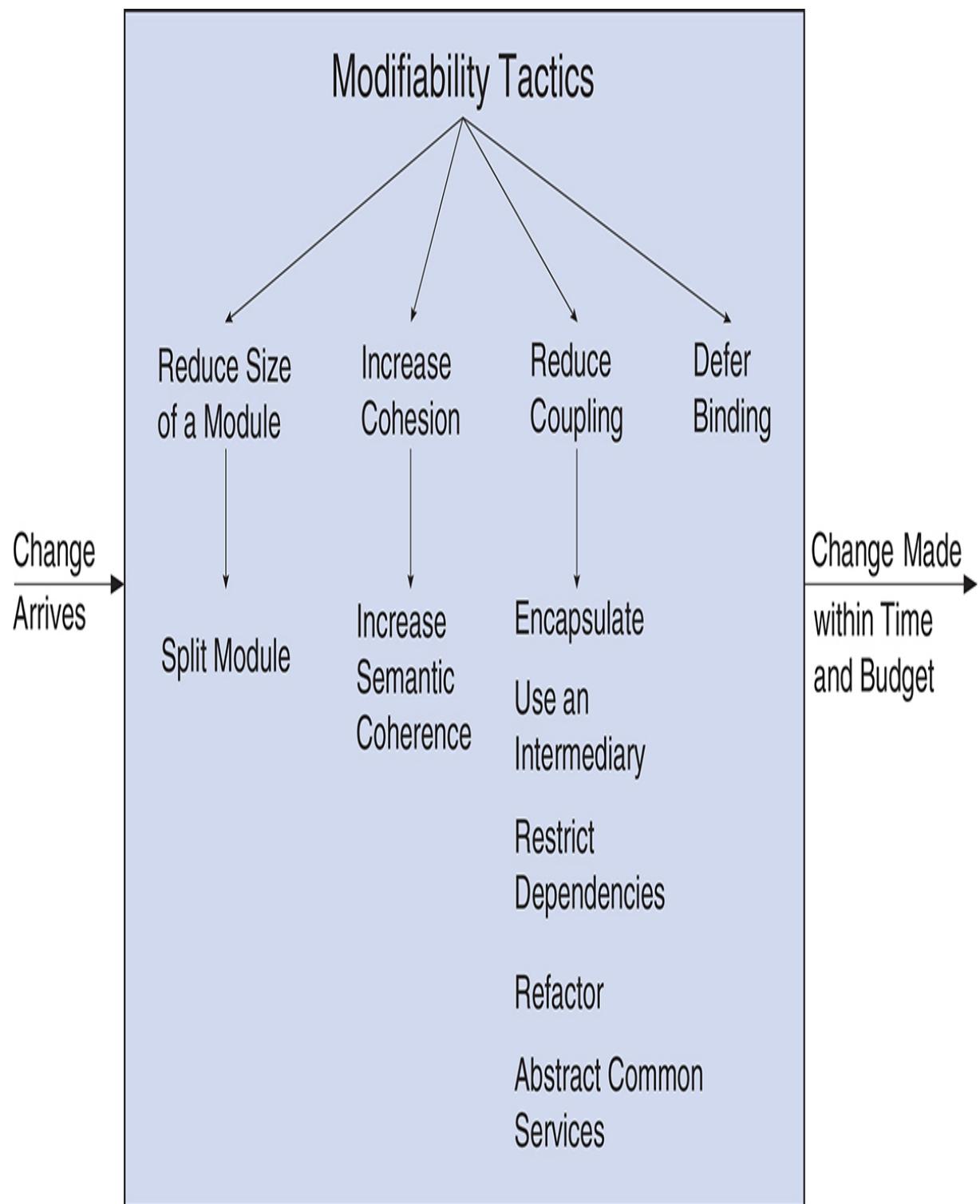


FIGURE 3.7 Modifiability Tactics Categorization

Within the Increase Cohesion category, the tactics are:

- *Split Module*. If the module being modified includes responsibilities that are not cohesive, modification costs will likely be high. Refactoring the module to separate the responsibilities should reduce the average cost of future changes.
- *Redistribute responsibilities*. If (related) responsibilities A, A', and A'' are sprinkled across several distinct modules, they should be grouped together into a single module.

Within the Reduce Coupling category, the tactics are:

- *Encapsulate*. Encapsulation introduces an explicit interface to an element, hiding its implementation details. This ensures that changes to the implementation do not affect the clients, as long as the interface remains stable. All access to the element must then pass through this interface.
- *Use an Intermediary*. Intermediaries, such as Proxies, Bridges or Adapters, are used for breaking dependencies between a set of components Cⁱ or between Cⁱ and the system S.
- *Abstract Common Services*. Where multiple elements provide services that are similar, it may be useful to hide them behind a common abstraction. The resulting encapsulation hides the details of the elements from other components in the system.
- *Restrict dependencies*. This tactic restricts which modules a given module interacts with or depends on.

And within the Defer Binding category, we distinguish the tactics according to how late in the life-cycle functionality is bound to the system:

- *These tactics can be used to bind values at compile or build time:* Component replacement, Compile-time parameterization, Aspects.
- *These tactics bind values at deployment, startup, or initialization time:* Configuration-time binding, Resource files.
- *Tactics to bind values at runtime include:* Discovery, Interpret parameters, Shared repositories, Polymorphism.

3.5.2 Modifiability Patterns

In [section 3.2.3](#) we said that patterns differ from tactics in that tactics typically have just a single goal—the control of a quality attribute response—whereas patterns often attempt to achieve multiple goals and balance multiple competing forces. You might not even think of the some of the patterns that we present for modifiability as modifiability patterns at all. For example, we present the client-server pattern in this section. But first we begin with the uber modifiability pattern—layers.

3.5.2.1 Layered Pattern

In complex systems there is a need to develop and evolve portions independently. For this reason, developers need a clear and well-documented separation of concerns, so that modules may be independently developed and maintained. The software therefore needs to be segmented in such a way that the modules can be evolved separately with little interaction among the parts. To achieve this separation of concerns, we divide the software into units called *layers*. Each layer is a grouping of modules that offers a cohesive set of services. Layers completely partition the software, and each partition is exposed through a public interface.

There is, ideally, a unidirectional *allowed-to-use* relation among the layers.

But layering has its costs and tradeoffs. The addition of layers adds up-front effort and complexity to a system. And layers add run-time overhead; there is a performance penalty to every interaction that crosses multiple layers. In section 5.2.3 we will see how this pattern may be realized in terms of layered APIs.

3.5.2.2 Strategy

The strategy pattern allows the selection of an algorithm, typically from a family of related algorithms, at runtime.

This pattern is a realization of the Defer Binding tactic, and makes it easy to define a rich set of behaviors or policies and switch between them, typically at runtime, as needed. In doing so the code is not cluttered with complex switch or if statements to provide for the various options. This also eases the burden for programmers who might later want to evolve the system, adding even more options. The only drawback to this pattern is that it adds some up-front complexity and so it should not be used if there are a small and stable number of alternative algorithms or policies to be implemented.

3.5.2.3 Client-Server Pattern

The client-server pattern consists of a server providing services simultaneously to multiple distributed clients. The most common example is a web server providing information to multiple browser clients. You might not think of this pattern as a modifiability pattern at all. But patterns are complex and they can serve multiple purposes. There are, in fact, many benefits to the use of this pattern:

- The connection between a server and its clients is established dynamically.
- There is no coupling among the clients.
- The number of clients can easily scale and is constrained only by the capacity of the server.
- Server functionality can be scaled if needed with no impact on the clients.
- Clients and servers can evolve independently as long as their APIs remain stable.
- Common services can be shared among multiple clients.
- For interactive systems, the interaction with a user is isolated to the client.

Not all of these are benefits for modifiability, but certainly the aspects of this pattern that limit coupling and defer binding are obvious instantiations of modifiability tactics. A server typically has no prior knowledge of its clients.

The tradeoffs that this pattern introduces are primarily about performance, availability, and security. This pattern is implemented such that communication occurs over a network. Messages may be delayed by network congestion, leading to degradation (or at least unpredictability) of performance. For clients that communicate with servers over a network shared by other applications, provisions must be made for security. And servers may be single points of failure.

3.6 Design Concepts to Support Security

Security is a measure of the system's ability to protect data and resources from unauthorized access while still providing legitimate access to authorized actors. The most common approaches to characterizing and analyzing security focus on three important characteristics: confidentiality, integrity, and availability (CIA):

- Confidentiality is the property that data or services are protected from unauthorized access.
- Integrity is the property that data or services are not subject to unauthorized manipulation.
- Availability is the property that the system will be there for legitimate use, consistent with its specifications.

The security tactics provide strategies for achieving confidentiality and integrity. We have already discussed availability.

3.6.1 Security Tactics

Secure facilities in the physical world permit only limited access to certain resources (e.g., by building walls, having locked doors and windows, and security checkpoints), have means of detecting intruders (e.g., requiring visitors to wear badges, using motion detectors), have deterrence mechanisms (e.g., armed guards, razor wire), have reaction mechanisms (e.g., automatic locking of doors), and have recovery mechanisms (e.g., off-site backups).

These strategies are all relevant to computer-based systems and so lead to our four categories of security tactics: *Detect Attacks*, *Resist Attacks*, *React to Attacks*, and *Recover from Attacks*, as shown in [Figure 3.8](#).

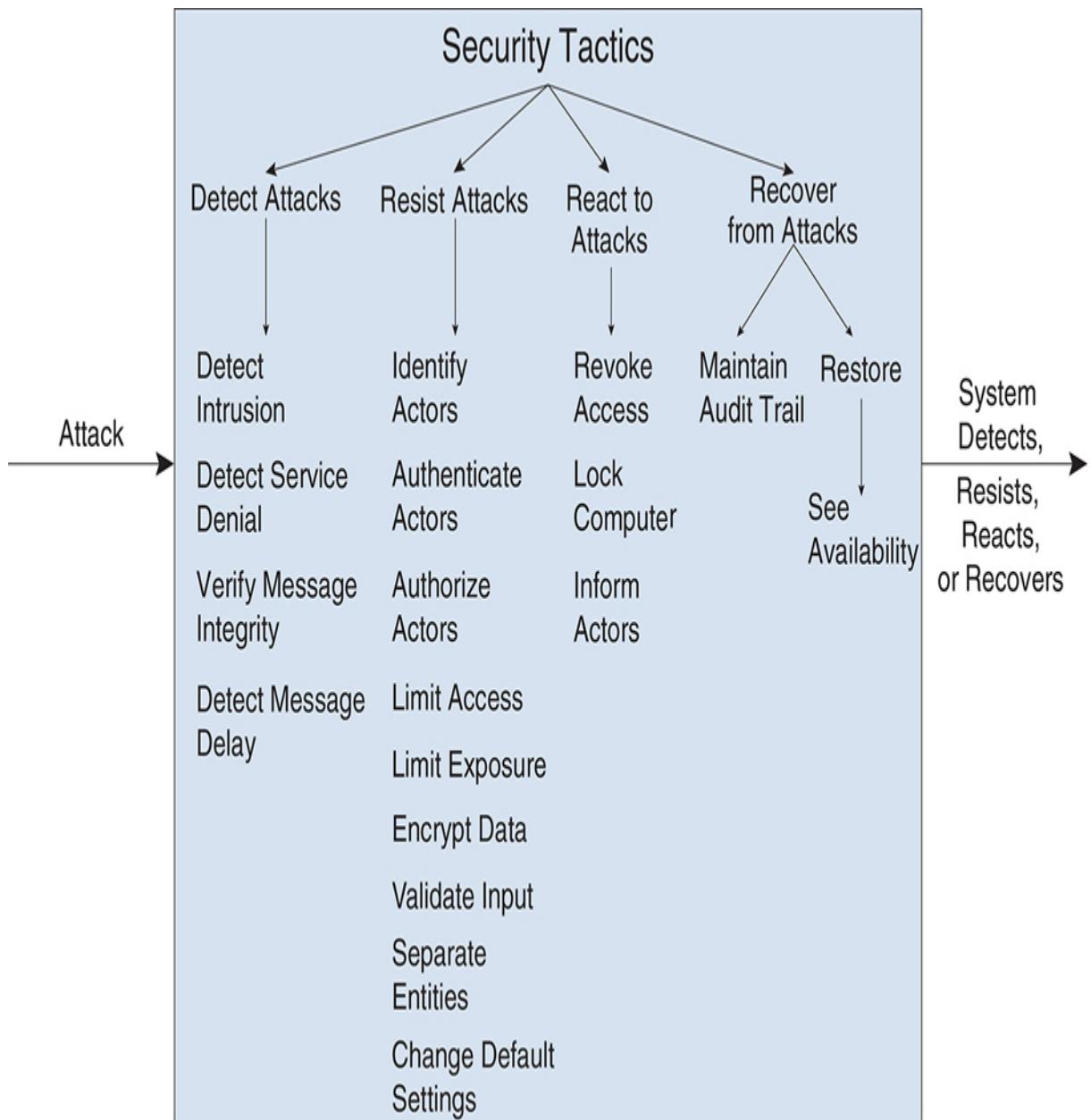


FIGURE 3.8 Security Tactics Categorization

Within the Detect Attacks category, the tactics are:

- *Detect intrusion.* This tactic compares network traffic or service request patterns within a system to a set of signatures or known patterns of malicious behavior stored in a database.

- *Detect service denial.* This tactic compares the pattern or signature of network traffic coming into a system to historical profiles of known denial-of-service (DoS) attacks.
- *Verify message integrity.* This tactic employs techniques such as checksums or hash values to verify the integrity of messages, resource files, deployment files, and configuration files.
- *Detect message delivery anomalies.* This tactic is used to detect man-in-the-middle attacks. If message delivery times are normally stable, then by checking the time that it takes to deliver or receive a message, it becomes possible to detect suspicious timing behavior. Similarly, abnormal numbers of connections and disconnections may indicate such an attack.

Within the Resist Attacks category, the tactics are:

- *Identify actors.* Identifying actors (users or remote computers) focuses on identifying the source of any external input to the system. Users are typically identified through user IDs. Other systems may be identified through access keys, IP addresses, protocols, ports, or some other means.
- *Authenticate actors.* Authentication means ensuring that an actor is actually who or what it purports to be. Passwords, digital certificates, two-factor authentication, and biometric identification provide means for authentication.
- *Authorize actors.* Authorization means ensuring that an authenticated actor has the rights to access and modify either data or services. This mechanism is usually enabled by providing some access control mechanisms within a system.

- *Limit access.* This tactic involves limiting access to computer resources. Limiting access might mean restricting the number of access points to resources, or restricting the type of traffic that can go through the access points. Both kinds of limits minimize the attack surface of a system.
- *Limit exposure.* This tactic focuses on minimizing the effects of damage caused by a hostile action. It is a passive defense since it does not proactively prevent attackers from doing harm. Limiting exposure is typically realized by reducing the amount of data or services that can be accessed through a single access point, and hence compromised in a single attack.
- *Encrypt data.* Confidentiality is usually achieved by applying some form of encryption to data and to communication. Encryption provides extra protection to persistently maintained or in-transit data beyond that available from authorization.
- *Separate entities.* Separating different entities limits the scope of an attack. Separation within the system can be done through physical separation on different servers attached to different networks, the use of virtual machines, or an “air gap”—that is, by having no electronic connection between different portions of a system.
- *Validate input.* Cleaning and checking input as it is received by a system, or portion of a system, is an important early line of defense in resisting attacks. This is often implemented by using a security framework or validation class to perform actions such as filtering, canonicalization, and sanitization of input.
- *Change credential settings.* Many systems have default security settings assigned when the system is delivered.

Forcing the user to change those settings will prevent attackers from gaining access to the system through settings that may be publicly available.

Within the React to Attacks category, the tactics are:

- *Revoke access*. If the system or an administrator believes that an attack is underway, then access can be limited to sensitive resources, even for normally legitimate users and uses.
- *Restrict login*. Repeated failed login attempts may indicate a potential attack. Many systems limit access from a particular computer if there are repeated failed attempts to access an account from that computer.
- *Inform actors*. Ongoing attacks may require action by operators, other personnel, or cooperating systems. Such personnel or systems—the set of relevant actors—must be notified when the system has detected an attack.

And, finally, within the Recover from Attacks category the tactics are:

- *Audit*. We audit systems—that is, keep a record of user and system actions and their effects—to help trace the actions of, and to identify, an attacker. We may analyze audit trails to attempt to prosecute attackers or to create better defenses in the future.
- *Nonrepudiation*. This tactic guarantees that the sender of a message cannot later deny having sent the message and that the recipient cannot deny having received the message.

In addition, all of the Availability tactics aid in recovering from attacks.

3.6.2 Security Patterns

There are many security patterns that have been developed over the years. Here we touch on just two of them, as examples: Intercepting Validator and Intrusion Detection.

3.6.2.1 Intercepting Validator

This pattern inserts a software element—an adapter—between the source and the destination of messages. This approach assumes greater importance when the source of the messages is outside the system.

The most common responsibility of this pattern is to implement the verify message integrity tactic, but it can also incorporate tactics such as detect intrusion and detect service denial, or detect message delivery anomalies.

The benefit of this pattern is that, depending on the specific validator that you create and deploy, this pattern can cover most of the waterfront of the “detect attack” category of tactics, all in one package.

The tradeoffs are that, as always, introducing an intermediary exacts a performance price.

Attack vectors change and evolve over time, so this component must be kept up-to-date so that it maintains its effectiveness. This imposes a maintenance obligation on the organization responsible for the system. But this maintenance cost must be paid in any case, if the system is to maintain its level of security.

3.6.2.1 Intrusion Detection

An intrusion prevention system (IPS) is a standalone element whose main purpose is to identify and analyze any suspicious activity. If the activity is deemed acceptable, it is

allowed. Conversely, if it is suspicious, the activity is prevented and reported. These systems often encompass most of the “detect attacks” and “react to attacks” tactics.

There are some tradeoffs associated with this pattern, however. For example, the patterns of activity that an IPS looks for change and evolve over time, so the patterns database must be constantly updated. Also, systems employing an IPS incur a performance cost. And IPSs are available as commercial off-the-shelf components, which might mean that they are not tailored for your specific application.

3.7 Design Concepts to Support Integrability

Software architects need to be concerned about more than just making separately developed components cooperate; they are also concerned with the costs and technical risks of integration tasks. These risks may be related to schedule, performance, or technology. Consider that a project needs to integrate a set of components C_1, C_2, \dots, C_n , into a system S . The task, then, is to design for, and analyze the costs and technical risks of, integrating additional, but not yet specified, components $\{C_{n+1}, \dots, C_m\}$. We assume we have control over S , but the development of the unspecified $\{C_i\}$ components may be outside our control.

Integration difficulty—the costs and the technical risks—can be thought of as a function of the size of and the “distance” between the interfaces of $\{C_i\}$ and S : Size is the number of potential dependencies between $\{C_i\}$ and S and distance is the difficulty of resolving differences in each of the dependencies. Distance may be any of: syntactic distance, data semantic distance, behavioral semantic distance, temporal distance, or even resource distance. While we may

not capture size and distance as precise metrics, this gives a frame of reference for thinking about integrability.

3.7.1 Integrability Tactics

The goals for the integrability tactics are to reduce the costs and risks of adding new components, reintegrating changed components, and integrating sets of components together to fulfill evolutionary requirements. These tactics help to reduce size and distance attributes.

There are three categories of integrability tactics: *Limit Dependencies*, *Adapt*, and *Coordinate*, as shown in [Figure 3.9](#).

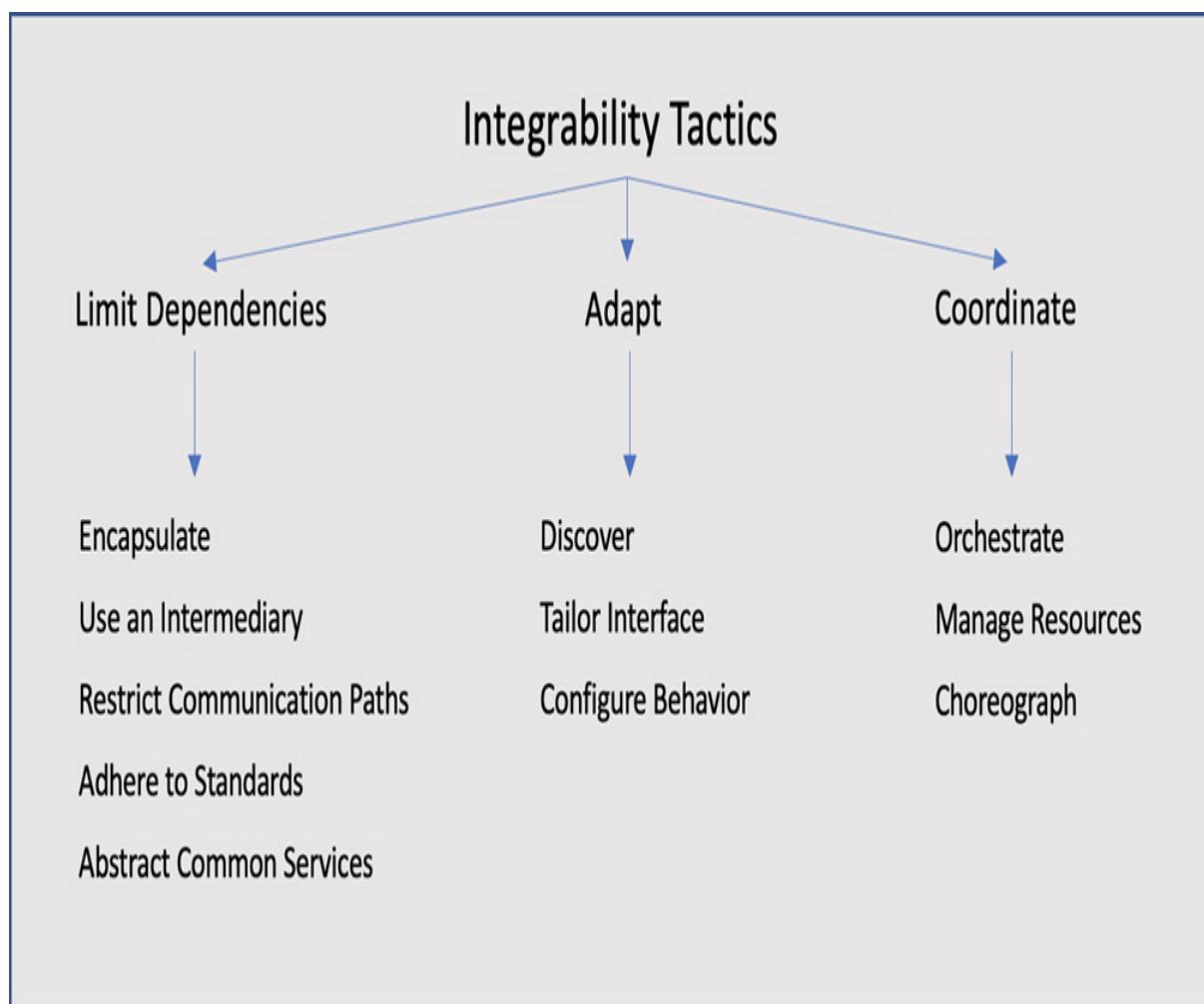


FIGURE 3.9 Integrability Tactics Categorization

The first category, Limit Dependencies, may remind you of the Reduce Coupling category from Modifiability. This is not surprising since limiting dependencies can be achieved, in large part, by reducing coupling. Within the Limit Dependencies category the tactics are:

- *Encapsulate.* Encapsulation introduces an explicit interface to an element and ensures that all access to the element passes through this interface. Dependencies on the element internals are eliminated, because all dependencies must flow through the interface.
- *Use an Intermediary.* Intermediaries are used for breaking dependencies between a set of components Ci or between Ci and the system S. Intermediaries can be used to resolve different types of dependencies (syntactic, behavior, data semantic, etc.).
- *Restrict Communication Paths.* This tactic restricts the set of elements with which a given element can communicate. In practice, this tactic is implemented by restricting an element's visibility (when developers cannot see an interface, they cannot employ it) and by authorization (i.e., restricting access to only authorized elements).
- *Adhere to Standards.* Standardization in system implementations is a primary enabler of integrability and interoperability, across both platforms and vendors. Some standards focus on defining syntax and data semantics. Others include richer descriptions, such as those describing protocols that include behavioral and temporal semantics.

- *Abstract Common Services*. Where multiple elements provide services that are similar, it may be useful to hide them behind a common abstraction. This abstraction might be realized as a common interface implemented by them all, or it might involve an intermediary that translates requests for the abstract service to more specific requests. The resulting encapsulation hides the details of the elements from other components in the system.

Within the Adapt category the tactics are:

- *Discover*. A discovery service is the mechanism by which applications and services locate each other. Entries in a discovery service are there because they were registered. This registration can happen statically, or it can happen dynamically when a service is instantiated.
- *Tailor Interface*. Tailoring an interface is a tactic that adds capabilities to, or hides capabilities in, an existing interface without changing the API or implementation. Capabilities such as translation, buffering, and data smoothing can be added to an interface without changing it.
- *Configure Behavior*. The behavior of a component can be configured during the build phase (recompile with a different flag), during system initialization (read a configuration file or fetch data from a database), or during runtime (specify a protocol version as part of your requests).

And, finally, within the Coordinate category the tactics are:

- *Orchestrate*. Orchestrate is a tactic that uses a control mechanism to coordinate and manage the invocation of services so that they can remain unaware of each other.

Orchestration helps with the integration of a set of loosely coupled reusable services to create a system that meets a new need.

- *Choreograph*. Choreography is an alternative control mechanism where a process is implemented as steps in a flow, each triggered by an event message upon completion of the previous step and unaware of the other steps in the flow. This results in composable, loosely coupled services without an orchestrator. Orchestration is centralized whereas Choreography is distributed.
- *Manage Resources*. A resource manager is a form of intermediary that governs access to computing resources; it is similar to the restrict communication paths tactic. With this tactic, software components are not allowed to directly access some computing resources (e.g., threads or blocks of memory), but instead request those resources from a manager.

3.7.2 Integrability Patterns

Many patterns exist to support integrability. Here we briefly describe just a few of them. The first three are related—adapters, bridges, and mediators.

3.7.2.1 Adapters, Bridges, and Mediators

An adapter is a form of encapsulation whereby some component is encased within an alternative abstraction. An adapter is the only element allowed to use that component; every other piece of software uses the component's services by going through the adapter. The adapter transforms the data or control information for the component it wraps.

A bridge translates some “requires” assumptions of one arbitrary component to some “provides” assumptions of another component. The key difference between a bridge and an adapter is that a bridge is independent of any particular component. Also, the bridge must be explicitly invoked by some external agent—possibly but not necessarily by one of the components the bridge spans.

Mediators exhibit properties of both bridges and adapters. The major distinction between bridges and mediators, is that mediators incorporate a planning function that results in runtime determination of the translation, whereas bridges establish this translation at bridge construction time.

All three of these patterns allow access to an element without forcing a change to the element or its interface. But these benefits, of course, come at a cost. Creating any of the patterns requires up-front development work. And all of the patterns will introduce some performance overhead while accessing the element, although typically this overhead is small.

And, as a side note, a fourth pattern with a similar intent is Facade. We will discuss an example of the Facade pattern in [Chapter 5](#), where it is instantiated as an API Gateway.

3.7.2.2 Services

Another common pattern used to ease integrability concerns is services. A service (whether “micro” or not) is an independent decoupled software component that can be developed, deployed and scaled independently. The services pattern describes a collection of distributed components that provide and consume services. Components have interfaces that describe the services they request and that they provide. A service’s QAs can be specified and guaranteed with a service level agreement (SLA).

Components perform their computations by requesting services from one another.

The benefits of the services pattern are that services are designed to be used by a variety of clients, leading them to be more generic. And services are independent; the only method for accessing a service is through its interface and through messages over a network. This means that they typically have loose coupling with other services and with their environment. Finally, services can be implemented heterogeneously, using whatever languages and technologies are most appropriate.

However, service-based architectures, because of their heterogeneity and distinct ownership, come with many interoperability mechanisms such as WSDL and SOAP. This adds some up-front complexity and overhead. (We will discuss APIs and API-centric design in [Chapter 5](#).)

3.7.2.3 Dynamic Discovery

Dynamic discovery applies the discover tactic to enable finding service providers at runtime. A dynamic discovery capability sets the expectation that the system will advertise the services available for integration and the information that will be available for each service.

The main benefit of this pattern is that it allows for flexibility in binding services together into a cooperating whole. For example, services may be chosen at startup or dynamically at runtime based on their pricing or availability or other properties. The cost of this flexibility is that dynamic discovery registration and de-registration must be automated, and tools for this purpose must be acquired or generated.

3.8 Summary

This chapter, although already quite long, has barely scratched the surface of making design decisions. But it has given principles for making design decisions—a way to reason rigorously about decision-making. And the chapter has provided several examples of design concepts (patterns and tactics) which are the building blocks of design. We hope that you can now examine other quality attributes, or other patterns for the QAs discussed here in exactly the same way as what we have shown. The ones we presented here are merely the most common ones, but they are not special in any other way.

In addition, technologies (externally developed components) are also design concepts which we did not discuss extensively here. In subsequent chapters of this book (for example, [Chapter 7](#) where we discuss cloud-based solutions) other specific design concerns such as technology choices will be presented.

This is all good news for the architect. Design, it turns out, is not mysterious. The design decisions that you can make for quality attributes comprise a substantial body of knowledge but this body of knowledge is tractable, and it is learnable.

3.9 Further Reading

The nine decision-making principles presented here were first described in A. Tang, R. Kazman, “Decision-Making Principles for Better Software Design Decisions”, *IEEE Software*, 38, Nov-Dec. 2021.

There are many references for design and architecture patterns. We already enumerated a few of these in [chapter 2](#). Among the best known are the “original” design patterns book: *Design Patterns: Elements of*

Reusable Object-Oriented Software by E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Addison-Wesley, 1994 and the book *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, by F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Wiley, 1996. This book was the beginning of a series that eventually totaled five volumes. But the set of patterns is never complete; new patterns continue to be created all the time. For example you can find a catalog of microservice patterns here:
<https://microservices.io/patterns/>

You can read about the tactics presented here, and other tactics, in greater detail, in *Software Architecture in Practice*, 4th edition, by L. Bass, P. Clements, and R. Kazman, Addison-Wesley, 2021.

The SWEBOK summarizes a set of nine software design principles: Hironori Washizaki, eds., Guide to the Software Engineering Body of Knowledge (SWEBOK), Version 4.0, IEEE Computer Society, 2024;
www.swebok.org.

There are several online catalogs of reference architectures developed by different cloud providers, they can be found at the following links:

- <https://learn.microsoft.com/en-us/azure/architecture/browse/>
- <https://aws.amazon.com/architecture/>
- <https://cloud.google.com/architecture>

3.10 Discussion Questions

1. Humans are all biased. How do the nine decision-making principles help to combat bias? How might you use these principles in practice?

2. If you had to design an architecture for a quality attribute driver that was not listed in this chapter, how would you go about determining (and validating) the tactics and patterns, including their costs, benefits, and tradeoffs?
3. How could you use a tactic to modify and improve a design pattern? Give a concrete example.
4. What is the relationship between tactics, patterns, and externally developed components? How can you use this knowledge to select and analyze components? Give an example.
5. Tactics are foundational, but the categorization can never be said to be “done”. New designs and design approaches do emerge over time. Can you identify a tactic that was not included in the categorizations presented here?

4. The Architecture Design Process

In this chapter we provide a detailed discussion of ADD, the design method that is at the heart of this book. We begin with an overview of the method and of each of its steps. This overview is followed by more detailed discussions of different aspects that need to be considered when performing these steps. We also discuss the identification and selection of design concepts, the production of structures from these design concepts, the definition of interfaces, the production of preliminary documentation, and, finally, a technique to track design progress.

Why read this chapter?

Architectural design is the foundation for any complex software system but it is most often done in an *ad hoc* way, if at all. It is our belief, and our experience, that having a structured way of doing design results in better, more predictable, outcomes. In this chapter we provide you a method that allows design to be performed in a structured way and we hope to convince you that this truly leads to better outcomes.

4.1 The Need for a Principled Method

In [Chapter 2](#), we discussed the various concepts associated with design. The question is, how do you actually perform design? Performing design to *ensure* that the drivers are satisfied requires a principled method. By “principled,” we refer to a method that takes into account all of the relevant aspects that are needed to produce an adequate design. Such a method provides guidance that is necessary to guarantee that your drivers are satisfied. To achieve this goal in a cost-effective, repeatable way, you need a method that guides you in combining and incorporating reusable design concepts.

Performing design adequately is important because architecture design decisions have significant consequences at different points in a project’s lifetime. For example, during an estimation phase, an appropriate design will allow for a better estimation of cost, scope, and schedule. During development, an appropriate design will be helpful to avoid later rework and facilitate development and deployment. Finally, a clear understanding of what architectural design involves is necessary to better manage aspects of technical debt.

4.2 Attribute-Driven Design 3.0

Architecture design is performed in a series of *rounds* across the development of a software project. Each design round may take place within a project increment such as a sprint. Within these rounds, a number of *design iterations* is performed. Perhaps the most important characteristic of the ADD method is that it provides detailed, step-by-step guidance on the tasks that have to be performed inside the design iterations.

ADD has been used successfully for more than 20 years. Version 2.0 was published in 2006 and version 3.0 was published in 2016, in the first edition of this book. While there have been dramatic changes in the development landscape since the publication of version 3.0, the method itself remains unchanged. That attests to the general applicability of ADD.

We have added chapters in the book that explain how it can be used with the software development techniques that are widely used today, but the steps and artifacts of the method, which are shown in [figure 3.1](#) remain the same. This is good news for you as an architect!

In the following subsections we provide an overview of the activities in each of its steps. Note that we present them as linear steps, but there may be significant interaction between them, resulting in micro-iterations. This is the nature of design; it is just not linear.

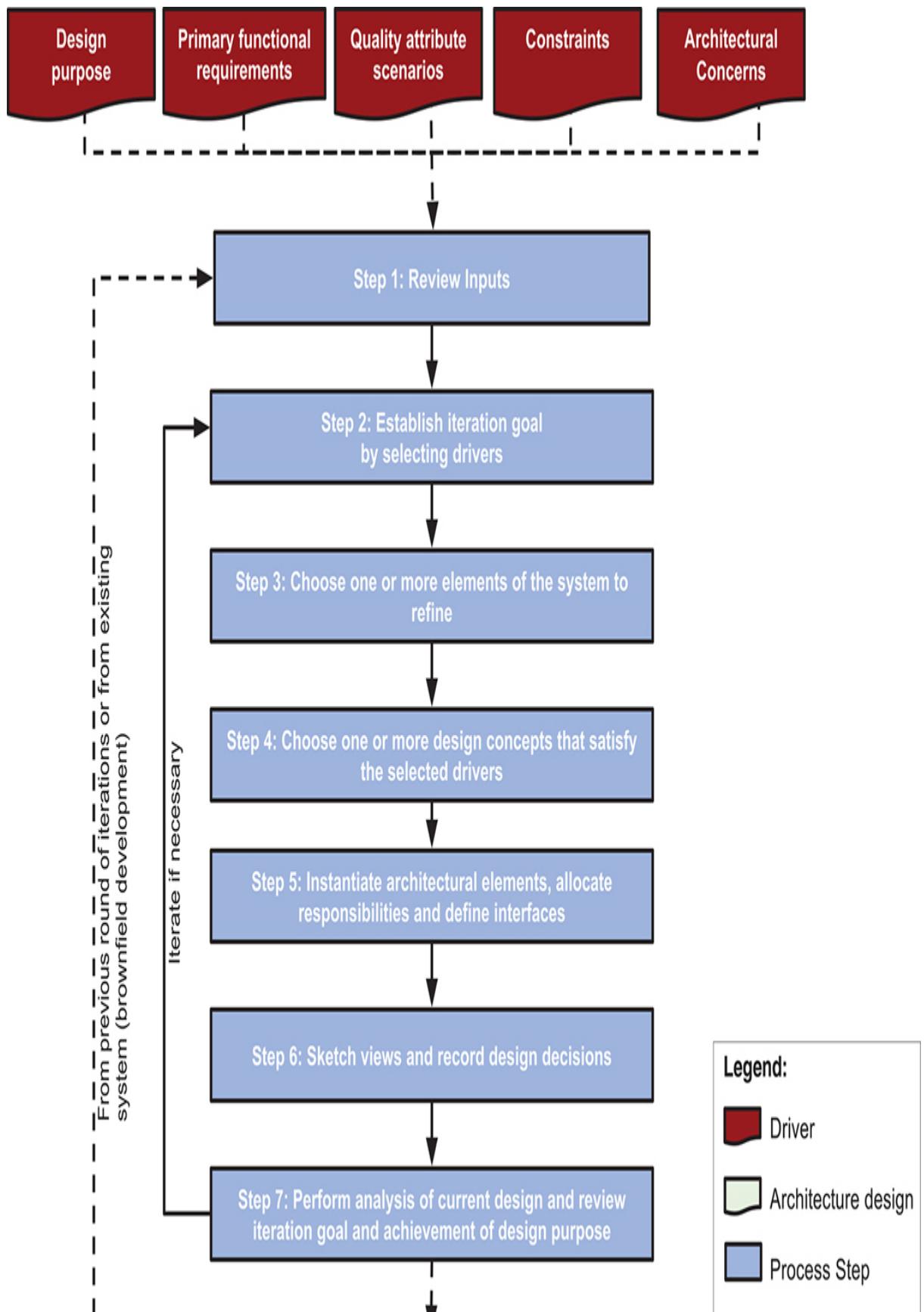




FIGURE 4.1 Steps and artifacts of ADD version 3.0

4.2.1 Step 1: Review Inputs

Before starting a design round, you need to make sure that the inputs to the design process are available and correct. First, you need to ensure that you are clear about the *purpose* for the design activities that will ensue. The purpose may be, for example, to produce a design for early estimation, to refine an existing design to build a new increment of the system, or to design and generate a prototype to mitigate certain technical risks (see [Section 2.4.1](#) for a discussion of the design purpose). Also, you need to make sure that the other drivers needed for the design activity are available. These include primary functional requirements, quality attribute scenarios, architectural constraints, and concerns. Finally, if this is not the first design round, or if this is not greenfield development, an additional input that you need to consider is the existing architecture design.

At this point, we assume that primary functionality and quality attribute scenarios have been prioritized, ideally by your most important project stakeholders. (If not, there are techniques that you can employ to elicit and prioritize them, as discussed in [Sections 2.4.2](#) and [2.4.3](#).) You, as the architect, must now “own” these drivers. You need to check, for example, whether any important stakeholders were overlooked in the original requirements elicitation process, or whether any business conditions have changed since the prioritization was performed. These drivers really do “drive”

design, so getting them right and getting their priority right is crucial. We cannot stress this point strongly enough. Software architecture design, like most activities in software engineering, is a “garbage in, garbage out” process. The results of ADD cannot be good if the inputs are poorly formed.

As a rule of thumb, you should be able to start designing if, besides the design purpose, constraints, and initial architectural concerns, you have established the primary use cases and the most important quality attribute scenarios. This, of course, does not mean you will make decisions only about these drivers: You still need to address other quality attribute scenarios, use cases and architectural concerns, but these can be treated later on.

The drivers become part of an architectural design backlog that you should use to perform the different design iterations. We discuss this idea in more depth in [Section 4.8.1](#).

4.2.2 Step 2: Establish the Iteration Goal by Selecting Drivers

A design round represents the architecture design activities performed within a development cycle if an iterative development model is used, or the whole set of architecture design activities if a waterfall model is used. Through one or more rounds, you produce an architecture that suits the established design purpose.

A design round is generally performed in a series of design iterations, where each iteration focuses on achieving a particular goal. Such a goal typically involves designing to satisfy a subset of the drivers. For example, an iteration goal could be to create structures from elements that will

support a particular performance scenario, or that will enable a use case to be achieved. For this reason, when performing design, you need to establish a goal before you start a particular design iteration.

Selecting an adequate goal for a design iteration can sometimes be challenging. The goal may be too small (a unique and relatively simple driver is selected) or too big (for example, too many drivers are selected). Some guidelines in establishing an iteration goal are the following:

- The goal may be to address a single important driver, for example, the architectural concern of decomposing the system or addressing a challenging quality attribute scenario.
- The goal may be to make decisions to satisfy a set of similar drivers, for example, making design decisions to address a family of use cases or related quality attribute scenarios.
- The goal may be to make decisions to satisfy a set of related drivers, for example, making design decisions to address a user story and an associated quality attribute scenario.

As we will discuss in [Section 4.3](#), depending on the type of system whose architecture is being designed, there may be a “best”—or at least strongly suggested—ordering of the iteration goals that need to be addressed. For example, for a greenfield system in a mature domain, your initial goal is typically to identify an overall structure for the system by choosing a reference architecture.

4.2.3 Step 3: Choose One or More Elements of the System to Refine

This step is where the core design activities start. Satisfying drivers requires you to produce one or more architectural structures. These structures are composed of interrelated elements, and those elements are generally obtained by refining other elements that you previously identified in an earlier iteration. Refinement can mean decomposition into finer-grained elements (top-down approach), combination of elements into coarser-grained elements (bottom-up approach), or improvement of previously identified elements. For greenfield development, you can start by establishing the system context and then selecting the only available element—that is, the system itself—for refinement by decomposition. For existing systems or for later design iterations in greenfield systems, you normally choose to refine elements that were identified in prior iterations.

The elements that you will select are the ones that are involved in the satisfaction of specific drivers. For this reason, when design is performed for an existing system, you need to have a good understanding of the elements that are part of the as-built architecture of the system. This may involve some “detective work,” reverse engineering, or discussions with developers.

We have presented steps 2 and 3 in the order they appear in the method. That is to say, step 2 precedes step 3. However, in some cases you may need to reverse this order. For example, when designing a greenfield system or when fleshing out certain types of reference architectures, you will, at least in the early stages of design, focus on elements of the system and start the iteration by selecting a particular element and then consider the drivers that you want to address.

4.2.4 Step 4: Choose One or More Design Concepts That Satisfy the Selected Drivers

Choosing the design concepts is probably the most difficult decision you will face in the design process, because it requires you to identify alternatives among design concepts that can be used to achieve your iteration goal, and to make a selection from these alternatives. As we saw in [Section 2.5](#), different types of design concepts exist, and, for each type, there may be many options. This can result in a considerable number of alternatives that need to be analyzed to make a choice, though the existence of architecture standards may help to constrain the number of viable alternatives. In [Section 4.4](#), we discuss the identification and selection of design concepts in more detail.

4.2.5 Step 5: Instantiate Architectural Elements, Allocate Responsibilities, and Define Interfaces

Once you have selected one or more design concepts, you must make another design decision, which involves *instantiating* elements out of the design concepts that you selected. Instantiation means adjusting the selected design concepts to the problem at hand. For example, if you selected the Layers pattern as a design concept, you must decide how many layers will be used, since the pattern itself does not prescribe a specific number. In this example, the layers are the elements that are instantiated. If you selected a particular availability tactic, such as “Redundant spare”, instantiation implies deciding how this particular tactic will be achieved, for example by adding replication to a

particular element. In certain cases, instantiation can mean configuration. For example, you may have dedicated an iteration to selecting technologies and associating them with the elements in your design. In further iterations, you might refine these elements by making finer-grained decisions about how they should be configured to support a particular driver, such as a quality attribute.

After instantiating the elements, you need to allocate responsibilities to each of them. For example, in a typical backend of a web-based enterprise system, at least three layers are usually present: the API layer, the business layer, and the data layer. The responsibilities of these layers differ: The responsibilities of the API layer include exposing endpoints, whereas the responsibilities of the data layer include managing the persistence of data. An additional utility layer may also be considered.

Instantiating elements is just one of the tasks you need to perform to create structures that satisfy a driver or a concern. The elements that have been instantiated also need to be connected, to allow them to collaborate with one another. This requires the existence of *relationships* between the elements and the exchange of information through some kind of interface. The interface is a contractual specification of how information should flow between the elements. [Section 3.5](#) provides more details on how the different types of design concepts are instantiated and how structures are created, and [Section 3.6](#) discusses how interfaces can be defined.

The activities that are performed in this step are frequently performed by creating some type of diagram, either on a whiteboard or using some diagramming tool. Finally, it should be noted there may be a lot of back and forth (micro-iterations) among steps 3 and 5, as these are the central steps of the process where design decisions are made.

4.2.6 Step 6: Sketch Views and Record Design Decisions

At this point, you have finished performing the design activities for the iteration. Nevertheless, you may not have taken any actions to ensure that the views—the representations of the structures you created—are preserved. For instance, if you performed the previous step in a conference room, you probably ended up with a series of diagrams on a whiteboard. This information is essential, and you need to capture it so that you can later analyze and communicate it to other stakeholders.

The views that you have created are almost certainly incomplete, so these diagrams may need to be revisited and refined in a subsequent iteration. This is typically done to accommodate elements resulting from other design decisions that you will make to support additional drivers. This factor explains why we speak of “sketching” the views in ADD—that is, creating a preliminary type of documentation. The more formal, more fully fleshed-out documentation of these views—should you choose to produce them—occurs only after a number of design iterations have been finished (as part of the architectural documentation activity discussed in [Section 1.2.2](#)).

In addition to storing the sketches of the views, you should record the significant decisions that are made in the design iteration, and the reasons that led to these decisions (i.e., the rationale), to facilitate later analysis and understanding of the decisions. For example, decisions about important tradeoffs might be recorded at this time. During a design iteration, decisions are primarily made in steps 4 and 5. [Section 4.7](#) provides further information on how to create preliminary documentation *during* design, including creating sketches, recording design decisions and their rationale.

4.2.7 Step 7: Perform Analysis of Current Design and Review Iteration Goal and Achievement of Design Purpose

By the time you reach step 7, you should have created a partial design that addresses the goal established for the iteration. Making sure that this is actually the case is a good idea, so as to avoid unhappy stakeholders and later rework. You can perform the analysis yourself by reviewing the sketches of the views and design decisions that you recorded, but an even better idea is to have someone else help you review this design. We do this for the same reason that organizations frequently have a separate testing/quality assurance group: Another person will not share your assumptions, and will have a different experience base and a different perspective. Pulling in someone with a different point of view can help you find flaws, in both code and architecture. We discuss analysis in more depth in [Chapter 11](#).

Once the design performed in the iteration has been analyzed, you should review the state of your architecture in terms of the established design purpose. This means considering if, at this point, you have performed enough design iterations to satisfy the drivers that are associated with the design round as well as considering whether the design purpose has been achieved or if additional design rounds are needed in future project increments. [Section 4.8](#) describes simple techniques that allow you to keep track of design progress.

4.2.8 Iterate If Necessary

Ideally, you should perform additional iterations and repeat steps 2 to 7 for every driver that was considered as part of the input. More often than not, such iterations are not possible because of time or resource constraints that force you to stop the design activities and move on to the next activities in the development process—typically implementation.

What are the criteria for evaluating if more design iterations are necessary? We let *risk* be our guide. You should at least have addressed the drivers with the highest priorities. Ideally, you should have assured that critical drivers are satisfied or, at least, that the design is “good enough” to satisfy them. Finally, when performing iterative development, you can choose to perform one design round in every project iteration. The initial rounds should focus on addressing the drivers with the highest priority, while subsequent rounds focus on making design decisions for other drivers with less priority or on new drivers that appear as development progresses.

4.3 Applying ADD to different system contexts

When writing a paper or an essay, you might have experienced the much-dreaded “fear of the blank page”. Similarly, when you start designing an architecture, you may face a situation in which you ask yourself, “How do I begin designing?” To answer this question, you need to consider which type of system you are designing.

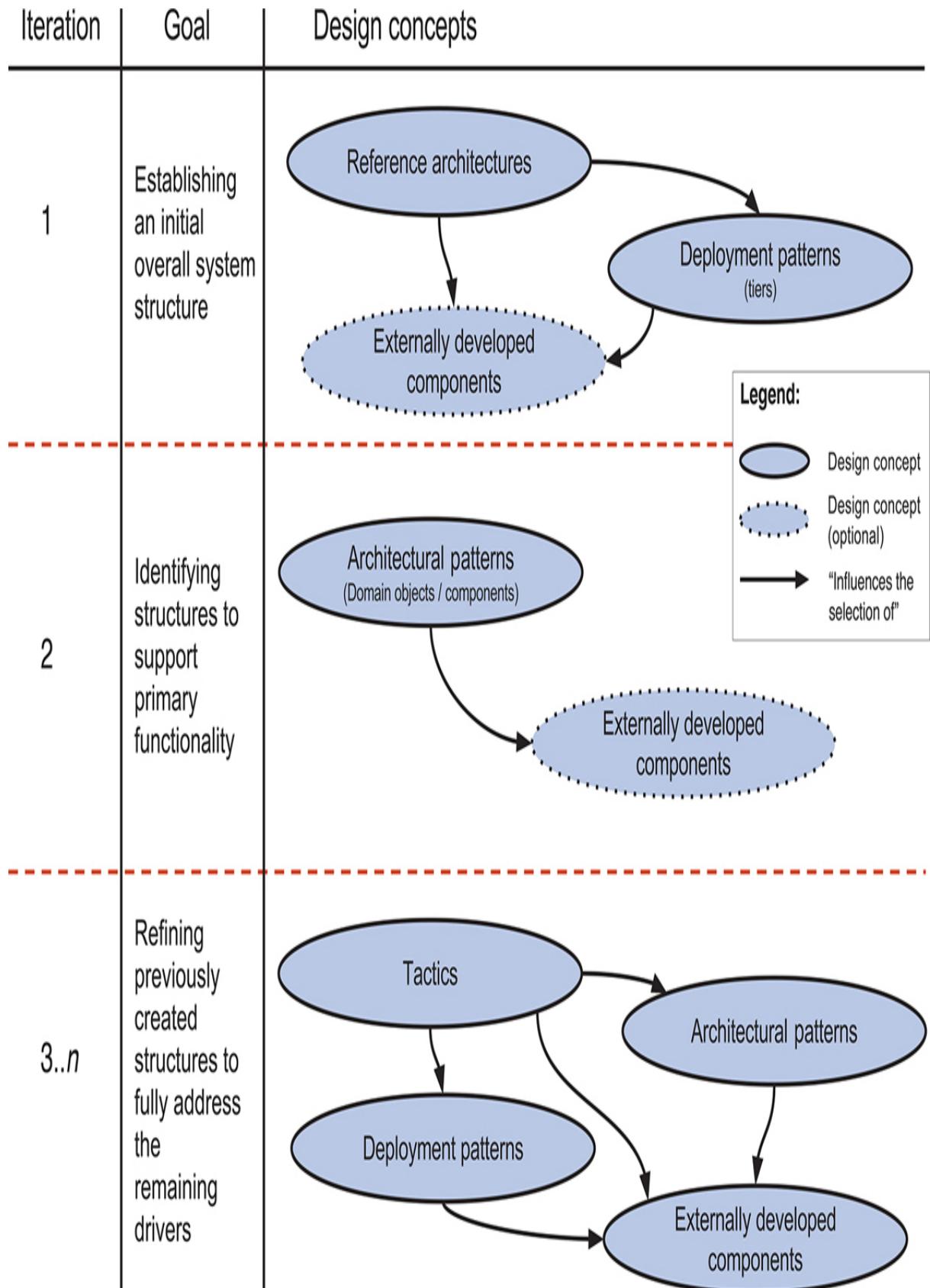
Design of software systems falls into three broad categories: (1) the design of a *greenfield* system for a mature (i.e., well-known) domain; (2) the design of a greenfield system for a domain that is novel (i.e., a domain that has a less established infrastructure and knowledge base); and (3) the

design for making changes to an existing system (*brownfield*). Each one of these categories involves different uses of ADD, and different ways that ADD can be applied to guide the design process.

4.3.1 Design of Greenfield Systems for Mature Domains

The design of a greenfield system for a mature domain occurs when you are designing an architecture for a system that is built from “scratch” and when this type of system is well known and understood—that is, when there is an established portfolio of tools and technologies, and an associated knowledge base. Examples of mature domains include the following:

- Traditional desktop applications
- Interactive applications that run on a mobile device
- Enterprise applications accessed from a web browser, which store information in a relational database, and which provide support for partially or fully automating business processes
- A microservice that manages a single domain entity



Legend:

- Design concept (Solid blue oval)
- Design concept (optional) (Dotted blue oval)
- "Influences the selection of"

FIGURE 4.2 Design concept selection roadmap for greenfield systems

Because these types of applications are relatively common, some general architectural concerns associated with their design are well known, well supported, and well documented. If you are designing a new system that falls into this category, we recommend the following roadmap (shown in [Figure 4.2](#)).

The goal of your **initial** design iteration(s) should be to address the general architectural concern of establishing an initial overall system structure. Is this to be a three-tier client-server application, a peer-to-peer application, a mobile app connecting to a Big Data back-end, and so on? Each of these options will lead you to different architectural solutions, and these solutions will help you to achieve your drivers. To achieve this iteration goal, you will select some design concepts. Specifically, you will typically choose one or more reference architectures and patterns, including patterns for deployment of the system (see [Section 6.2](#)). You may also select some externally developed components, such as frameworks. The types of frameworks that are typically chosen in early iterations are either “full-stack” frameworks that are associated with the selected reference architectures, or more specific frameworks that are associated with elements established by the reference architecture (see [Section 3.2.1](#)). In this first iteration, you should review all of your drivers to select the design concepts, but you will probably pay more attention to the constraints and to quality attributes that are not associated with specific functionalities and that favor particular reference architectures or require particular deployment configurations. Consider an example: If you select a reference architecture for Big Data systems, you have presumably chosen a quality attribute such as low latency

with high data volumes as your most important driver. Of course, you will make many subsequent decisions to flesh out this early choice, but this driver has already exerted a great influence on your design such as the selection of a particular reference architecture.

The goal of your **next** design iteration(s) should be to identify structures that support the primary functionality. As noted in [Section 2.4.3](#), allocation of functionality (i.e., use cases or user stories) to elements is an important part of architectural design because it has critical downstream implications for modifiability and allocation of work to teams. Furthermore, once functionality has been allocated, the elements that support it can be refined in later iterations to support the quality attributes associated with these functionalities. For example, a performance scenario may be associated with a particular use case. Achieving the performance goal may require making design decisions across all of the elements that participate in the achievement of this use case. To allocate functionality, you usually refine the elements that are associated with the reference architecture by decomposing them. A particular use case may require the identification of multiple elements. For example, if you have selected a web application reference architecture, supporting a use case will probably require you to identify modules across the different layers associated with this reference architecture. Finally, at this point you should also be thinking about allocating functionality—associated with modules—to (teams of) developers.

The goal of your **subsequent** design iterations should be to refine the structures you have previously created to fully address the remaining drivers. Addressing these drivers, and especially quality attributes, will likely require you to use the three major categories of design concepts—tactics,

patterns, and externally developed components such as frameworks—as well as commonly accepted design best practices such as modularity, low coupling, and high cohesion. For example, to (partially) satisfy a performance requirement for the search use case in a web application, you might select the “maintain multiple copies of data” tactic and implement this tactic by configuring a cache in a framework that is used inside an element responsible for persisting data.

This roadmap is appropriate for the initial project iterations, but it is also extremely useful for early project estimation activities (see the discussion about the architecture design process during pre-sales in [Section 12.1.1](#)). Why have we created such a roadmap? First, because the process of starting an architectural design is always complex. Second, because many of the steps in this roadmap are frequently overlooked or done in an intuitive and ad hoc way, rather than in a well-considered, reflective way. Third, because different types of design concepts exist, and it is not always clear at which point in the design they should be used. This roadmap encapsulates best practices that we have observed in the most competent architecture organizations. Simply put, the use of a roadmap results in better architectures, particularly for less mature architects.

4.3.2 Design of Greenfield Systems for Novel Domains

In the case of novel domains, it is more challenging to establish a precise roadmap, because reference architectures may not exist and there may be few, if any, externally developed components that you can use. You are, more than likely, working from first principles and creating your own home-grown solutions. Even in this case, however,

general-purpose design concepts such as tactics and patterns can guide you, aided by strategic prototyping. In essence, your iteration goals will mostly be to continuously refine previously created structures to fully address the drivers.

Many times, your design goal will focus on the creation of prototypes so that you can explore possible solutions to the challenge that you are facing. In particular, you may need to focus on quality attributes and design challenges oriented toward issues such as performance, scalability, or security. We discuss the creation of prototypes in [Section 4.4.2](#).

Of course, the notion of “novel” is fluid. Mobile application development was a novel domain 15 or 20 years ago, but now it is a well-established field.

4.3.3 Design for an Existing System (Brownfield)

Architecture design for an existing system may occur for different purposes. The most obvious is maintenance—that is, when you need to satisfy new requirements or correct issues, and doing so requires changes to the architecture of an existing system. You may also be making architectural changes to an existing system for the purpose of *refactoring* (see [Chapter 10](#)). When refactoring, you change the architecture of an existing system, without altering its functions, to reduce technical debt, to introduce technology updates, or to fix quality attribute problems (e.g., the system is too slow, or insecure, or frequently crashes).

To be able to choose elements to decompose as part of the design process (step 3 of ADD), you need to first identify which elements are present in the architecture of the existing system. In this sense, before starting the design

iterations, your first goal should be to make sure that you have a clear understanding of the existing architecture of the system.

Once you understand the elements, properties, and relationships that constitute the architecture of the system, and the characteristics of the existing code base, you can perform design similar to what is done for greenfield systems after the initial design iteration. Your design iteration goals here will be to identify and refine structures to satisfy architectural drivers, including new functionality and quality attributes, and to address specific architectural concerns. These design iterations will typically not involve establishing a new overall system structure unless you are dealing with a major refactoring.

4.3.4 Designing to Replace a Legacy Application

As systems age, many of the technologies they were built upon become increasingly obsolete and their replacement is complex and expensive. Also, some of these systems have accumulated so much technical debt that its repayment becomes infeasible (see [Chapter 10](#)). When these situations occur, replacement of the system often becomes the only feasible choice. But complex systems are difficult to replace all at once; instead, a gradual replacement approach is often less risky and more palatable to management.

Patterns such as the Strangler Fig offer some help in this task. This pattern allows an existing application to be substituted gradually while it keeps operating. The application to be replaced is typically moved behind a proxy which initially delegates all of the service calls from clients to the legacy application. Parts of the legacy application then are redesigned and implemented, for example

following a microservices approach if the existing application is a monolith. Once the new parts are deployed, the proxy begins to route calls to them, without affecting existing clients. This approach continues gradually until all of the invocations that were originally processed by the legacy application are now processed by newly developed components. At that point, the legacy application has been “strangled” and can be removed from service; the replacement is complete. The name Strangler Fig was chosen as an analogy to its botanical counterpart where a fig plant slowly surrounds an existing tree which eventually dies once it is strangled.

When performing this gradual replacement, design iteration goals are focused on replacing one or more services provided by the legacy application. Each design iteration is focused on new architectural choices for those replacement services.

4.4 Identifying and Selecting Design Concepts

Freeman Dyson, the English physicist, once said the following: “A good scientist is a person with original ideas. A good engineer is a person who makes a design that works with as few original ideas as possible”. This quotation is particularly relevant in the context of software architecture design: Most of the time you don’t need to, and shouldn’t, reinvent the wheel. Rather, your major design activities are to identify and select design concepts to address the challenges and drivers that you encounter across the design iterations. Design is still an original and creative endeavor, but the creativity resides in the appropriate identification of these existing solutions and then on combining and adapting them to the problem at hand.

4.4.1 Identification of Design Concepts

The identification of design concepts can appear to be daunting, because of the vast number of design concepts that exist. There are likely dozens of design patterns, tactics and externally developed components that you could use to address any particular issue. To make things worse, these design concepts are scattered across many different sources: in the popular press, in research literature, in books, and on the Internet. Moreover, in many cases, there is no canonical definition of a concept. Different sites, for example, will define the Broker pattern in different, largely informal, ways. Finally, once you have identified the alternatives that might potentially help you achieve the design goals of the iteration, you need to select among them.

To identify which design concepts you need at a particular point, you should consider what we previously discussed regarding the design for different systems contexts. Different points in the design process usually require different types of design concepts. For example, when you are designing a greenfield system in a mature domain, the types of design concepts that will help you initially structure the system are reference architectures and deployment patterns. As you progress in the design process, you will use all of the categories of design concepts: tactics, architecture and design patterns, and externally developed components (see [chapter 3](#)). Keep in mind that to address a specific design problem, you can and often will use and combine different types of design concepts. For example, when addressing a security driver, you may employ a security pattern, a security tactic, a security framework, or some combination of these.

Once you have more clarity regarding the types of design concepts that you wish to use, you still need to identify alternatives—that is, design candidates. There are several ways to do so, although you will probably use a combination of these techniques rather than a single one:

- *Leverage existing best practices.* You can identify alternatives for your required design concepts by making use of catalogs that are available in printed or online form. Some design concepts, such as patterns, are extensively documented; others, such as externally developed components, are documented in a less thorough way. The benefits of this approach are that you can identify many alternatives, and that you can leverage the considerable knowledge and experience of others. The downsides are that searching for and studying the information can require a considerable amount of time, the quality of the documented knowledge is often unknown, and the assumptions and biases of the authors are unknown.
- *Leverage your own knowledge and experience.* If the system you are designing is similar to other systems you have designed in the past, you will probably want to begin with some of the design concepts that you have used before. The benefit of this approach is that the identification of alternatives is performed rapidly and confidently. The downside is that you may end up using the same ideas repeatedly, even if they are not the most appropriate for all the design problems that you are facing, and if they have been superseded by newer, better approaches. As the saying goes, “If you give a small child a hammer, all the world looks like a nail”.
- *Leverage the knowledge and experience of others.* As an architect, you have background and knowledge that you have gained through the years. This foundation

varies from person to person, especially if the types of design problems they have addressed in the past differ. You can leverage this information by performing the identification and selection of design concepts with some of your peers through brainstorming.

4.4.2 Selection of Design Concepts

Once you have identified a list of alternative design concepts, you need to select which one is the most appropriate to solve the design problem at hand. You can achieve this in a relatively simple way, by creating a table that lists the pros and cons associated with each alternative and selecting one of the alternatives based on those criteria and your drivers. [Table 4.1](#) shows a simplified example of such a table used to support the selection of different deployment paradigms.

Table 4.1 *Example of a Table to Support the Selection of Alternatives*

Name of Alternative	Pros	Cons
Monolithic application.	Easier to manage and deploy.	Updates require redeployment of the whole application. Horizontal scaling requires creating a replica of the entire application.
Microservices-based application.	Services can be replicated and updated independently.	More complex management.

You may also need to perform a more in-depth analysis to select the alternative. Methods such as CBAM (cost benefit analysis method) or SWOT (strengths, weaknesses, opportunities, threats) can help you to perform this analysis (see the sidebar “[The Cost Benefit Analysis Method](#)”).

The Cost Benefit Analysis Method

The CBAM is a method that guides the selection of design alternatives using a quantitative approach. This method considers that architectural strategies (i.e., combinations of design concepts) affect quality attribute responses, and that the level of each response in turn provides system stakeholders with some benefit called *utility*. Each architectural strategy provides a different level of utility, but also has a cost and takes time to implement. The idea

behind the CBAM is that by studying levels of utility and costs of implementation, particular architectural strategies can be selected based on their associated return on investment (ROI). The CBAM was conceived to be performed after an ATAM (architecture tradeoff analysis method), but it is possible to use the CBAM during design—that is, prior to the moment where the architectural evaluation is performed.

The CBAM takes as its input a collection of prioritized traditional quality attribute scenarios, which are then analyzed and refined with additional information. The addition is to consider several levels of response for each scenario:

- The worst-case scenario, which represents the minimum threshold at which a system must perform (utility = 0)
- The best-case scenario, which represents the level after which stakeholders foresee no further utility (utility = 100)
- The current scenario, which represents the level at which the system is already performing (the utility of the current scenario is estimated by stakeholders)
- The desired scenario, which represents the level of response that the stakeholders are hoping to achieve (the utility of the desired scenario is estimated by stakeholders)

Using these data points, we can draw a utility-response curve, as shown in [figure 4.3](#). After the utility-response curve is mapped for each of the different scenarios, a number of contemplated design alternatives may be considered, and their expected response values can be estimated. For example, if we are concerned about mean time to failure, we might consider three different

architectural strategies (i.e., redundancy options)—for example, no redundancy, cold spare, and hot spare. For each of these strategies, we could estimate their expected responses (i.e., their expected mean times to failure). In the graph shown here, the “e” represents one such option, placed on the curve based on its expected response measure.

Using these response estimates, the utility values of each architectural strategy can now be determined via interpolation, which provides its expected benefit. The costs of each architectural strategy are also elicited—one would expect hot spare to be the most costly, followed by cold spare and no redundancy.

Given all of this information, architectural strategies can now be selected based on their expected value for cost.

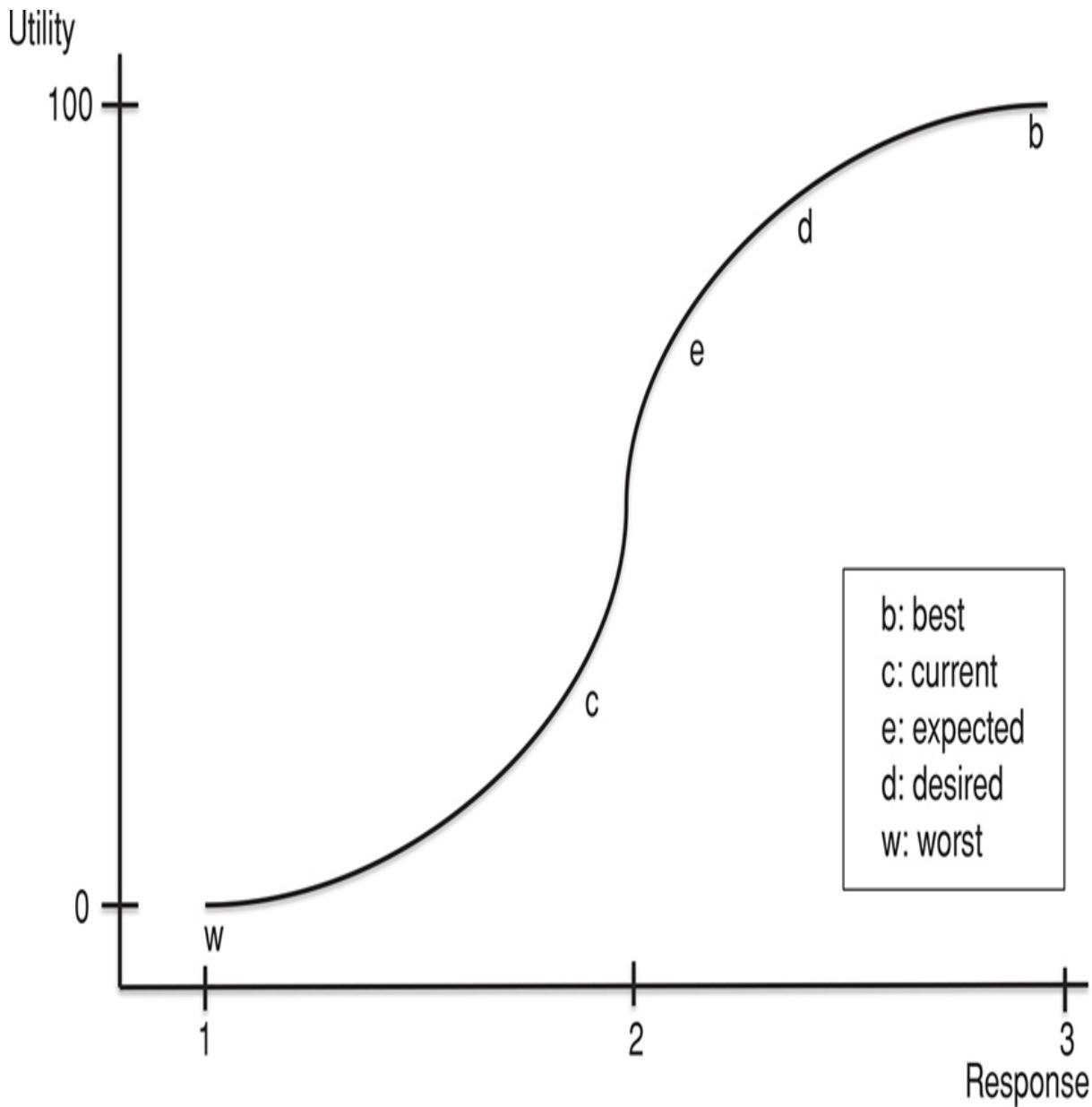


FIGURE 4.3 Utility-response curve

Although the CBAM may seem relatively complex and time-consuming at first, you need to consider that some design decisions can have enormous economic consequences—in terms of their costs, their benefits, and their effects on project schedule. You must decide if you are willing to take the chance of making these decisions solely using a gut-feeling approach versus this more rational and systematic approach.

In case the previous analysis techniques do not guide you to make an appropriate selection, you may need to create throwaway prototypes and collect measurements from them. The creation of early throwaway prototypes is a useful technique to help in the selection of externally developed components. This type of prototype is usually created in a “quick and dirty” fashion without too much consideration for maintainability or reuse. For these reasons, it is important to keep in mind that throwaway prototypes should not be used as a basis for further development.

Although the creation of prototypes can be costly compared to analysis (the ratio of costs is between 10 and 5 to 1, according to our sources), certain scenarios strongly motivate the creation of prototypes. Aspects that you should consider when deciding whether you will create a prototype include the following:

- Does the project incorporate emerging technologies?
- Is the technology new in the company?
- Are there certain drivers, particularly quality attributes, whose satisfaction using the selected technology presents risks (i.e., it is not understood if they can be satisfied)?
- Is there a lack of trusted information, internal or external, that provides some degree of certainty that the selected technology will be useful to satisfy the project drivers?
- Are there configuration options associated with the technology that need to be tested or understood?
- Is it unclear whether the selected technology can be integrated with other technologies that are used in the project?

If most of your answers to these questions are “yes,” then you should strongly consider the creation of a throwaway prototype.

When identifying and selecting design concepts, you need to keep in mind the *constraints* that are part of the architectural drivers, because some constraints will restrict you from selecting particular alternatives. For example, a constraint might require that all libraries and frameworks in the system do not use the GPL license; thus, even if you have found a framework that could be useful for your needs, you may need to discard it if it has a GPL license. Also, you need to keep in mind that the decisions regarding the selection of design concepts that you have made in previous iterations may restrict the design concepts that you can select in the future because of incompatibilities. For example, if you selected a web application reference architecture for use in an initial iteration, you cannot select a user interface framework intended for local applications in a subsequent iteration.

Finally, you need to remember that even though ADD provides guidance on how to perform the design process, it cannot ensure that you will make appropriate design decisions. Thorough reasoning and considering different alternatives (not just the first thing that comes to mind) are the best means to improve the odds of finding a good solution. We discuss doing “analysis in the design process” in [Chapter 11](#).

4.5 Producing Structures

Design concepts per se won’t help you satisfy your drivers unless you produce *structures*; that is, you need to identify and connect elements that are derived from the selected design concepts. This process is the *instantiation* of

architectural elements in ADD: creating elements and relationships between them, and associating responsibilities with these elements. It is important to remember that the architecture of a software system is composed of a set of structures, which can be grouped into three major categories:

- *Module structures*: composed of logical and static elements that exist at development time, such as files, modules, and classes
- *Component and connector (C&C) structures*: composed of dynamic elements that exist at runtime, such as processes and threads
- *Allocation structures*: composed of both software elements (from a module or C&C structure) and non-software elements that may exist both at development time and at runtime, such as file systems, infrastructure, and development teams

When you instantiate a design concept, you may actually produce more than one structure. For example, in a particular iteration you may instantiate the Layers pattern, which will result in a Module structure. As part of instantiating this pattern, you will need to choose the number of layers, their relationships, and the specific responsibilities of each layer. As part of the iteration, you may also study how a scenario is supported by the elements that you have just identified. For example, you could create instances of the logical elements in a C&C structure and model how they exchange messages (see [Section 4.6.1](#)). Finally, you may want to decide who will be responsible for implementing the modules inside each of the layers, which is an allocation decision.

4.5.1 Instantiating Elements

The instantiation of architectural elements depends on the type of design concept that you are working with:

- *Reference architectures.* In the case of reference architectures, instantiation typically means that you perform some sort of customization. As part of this work, you will add or remove elements that are part of the structure that is defined by the reference architecture. For example, if you are designing a microservice, you will probably need a service reference architecture that includes service, business, and data layers, as we showed in [Chapter 3](#).
- *Architectural and design patterns.* These patterns provide a generic structure composed of elements, their relationships and their responsibilities. As this structure is generic, you will need to adapt it to your specific problem. Instantiation usually involves transforming the generic structure defined by the pattern into a specific one that is adapted to the needs of the problem that you are solving. For example, consider the Pipe and Filters architectural pattern. It establishes the basic elements of computation—filters—and their relationships—pipes—but does not specify how many filters you should use for your problem or what their relationships should be. You will instantiate this pattern by defining how many pipes and filters are needed to solve your problem, by establishing the specific responsibilities of each of the filters, and by defining their topology.
- *Deployment patterns.* Similar to the case with architectural and design patterns, the instantiation of deployment patterns generally involves the identification and specification of infrastructure elements. If, for example, you are using a Load-Balanced Cluster pattern, instantiation may involve

identifying the number of replicas to be included in the cluster, the load-balancing algorithm, and the physical location of the replicas. When deploying to the cloud, instantiation of these patterns involves selecting resources offered by the chosen cloud provider.

- *Tactics*. This design concept does not prescribe a particular structure, so you will need to use other design concepts to instantiate a tactic. For example, you may select a security tactic of authenticating actors and instantiate it by creating a custom-coded ad hoc solution, or by using a security pattern, or by using an externally developed component such as a security framework.
- *Externally developed components*. The instantiation of these components may or may not imply the creation of new elements. For example, in the case of object-oriented frameworks, instantiation may require you to create specific classes that inherit from the base classes defined in the framework. This will result in new elements. Other approaches, which do not involve the creation of new elements, might include choosing a specific technology from a technology family that was identified in a previous iteration, associating a particular framework to elements that were identified in a previous iteration, or specifying configuration options for an element associated with a particular technology (such as a number of threads in a thread pool).

4.5.2 Associating Responsibilities and Identifying Properties

When you are creating elements by instantiating design concepts, you need to consider the responsibilities that are allocated to these elements. For example, if you instantiate

the Layers pattern and decide to use the traditional three-layer structure, you might decide that one of the layers will be responsible for managing the interactions with the users (typically known as the presentation layer). When instantiating elements and allocating responsibilities, you should keep in mind the high cohesion/low coupling design principle: Elements should have high cohesion (internally), defined by a narrow set of responsibilities, and low coupling (externally), defined by a lack of knowledge of the implementation details of other elements.

One additional aspect that you need to consider when instantiating design concepts is the properties of the elements. This may involve aspects such as the configuration options, statefulness, resource management, priority, or even hardware characteristics (if the elements that you created are physical nodes) of the chosen technologies. Identifying these properties supports analysis and the documentation of the design rationale.

4.5.3 Establishing Relationships Between the Elements

The creation of structures also requires making decisions with respect to the relationships that exist between the elements and their properties. Once again, consider the Layers pattern. You may decide that two layers are connected, but these layers will eventually be allocated to components that are, in turn, allocated to hardware. In such a case, you need to decide how communication will take place between these layers, as they have been allocated to components: Is the communication synchronous or asynchronous? Does it involve some type of network communication? Which type of protocol is used? How much information is transferred and at what rate? These design

decisions can have a significant impact with respect to achieving certain quality attributes such as performance.

4.6 Defining Interfaces

Interfaces are the externally visible properties of elements that establish a contractual specification that allows elements to collaborate and exchange information. There are two categories of interfaces: external and internal.

4.6.1 External Interfaces

External interfaces include application programming interfaces, or APIs, from other systems that are *required* by the system that you are developing and interfaces that are *provided* by your system to other systems. Required APIs are part of the constraints for your system, as you usually cannot influence their specification. [Chapter 5](#) provides an extensive discussion of APIs, including mechanisms for defining APIs which are necessary when your system is responsible for providing them.

Establishing a system context at the beginning of the design process is useful to identify external interfaces. This context can be represented using a system context diagram, as shown in [Figure 4.4](#). Given that external entities and the system under development interact via interfaces, there should be at least one external interface per external system (each relationship in the figure).

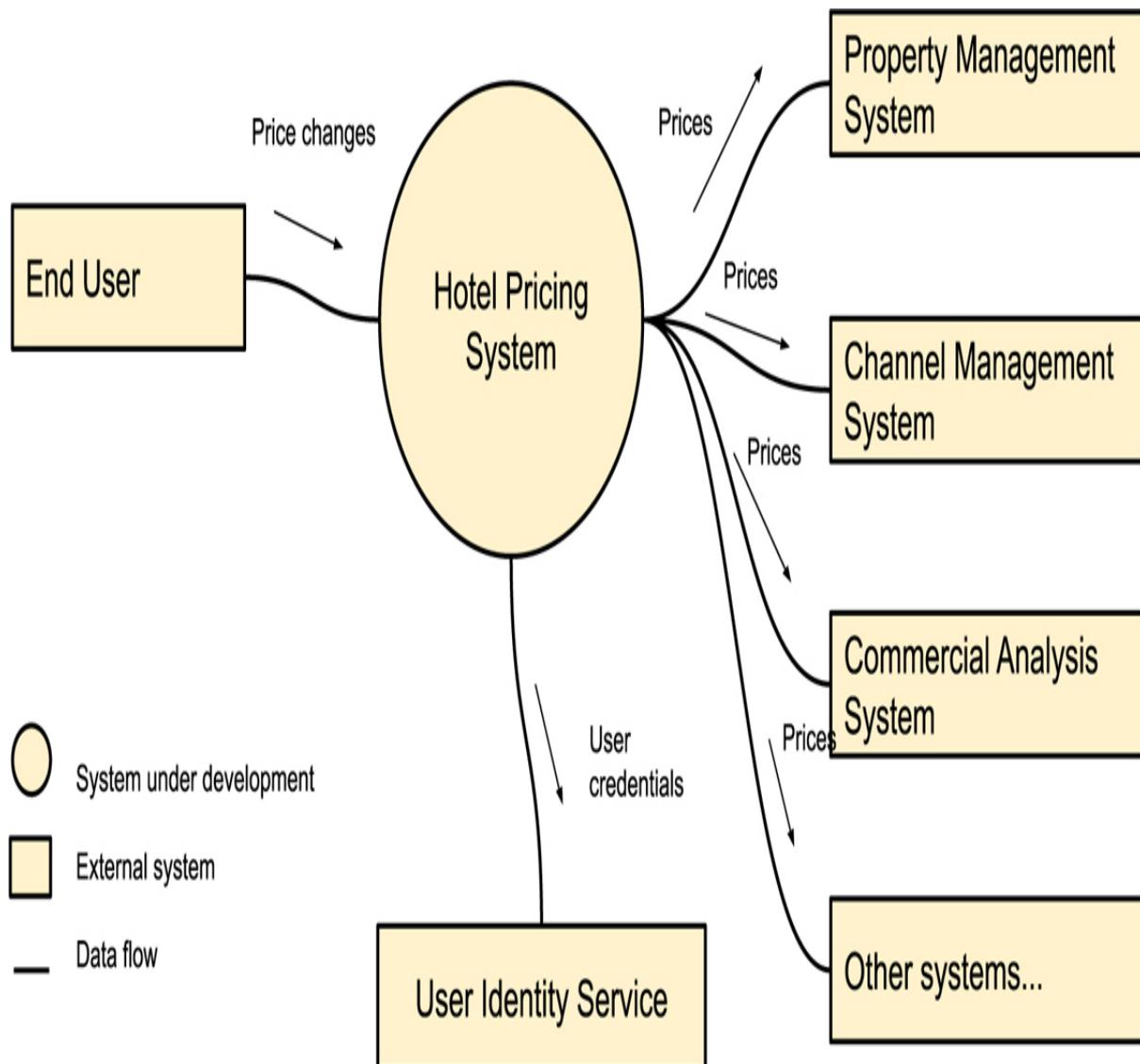


FIGURE 4.4 A system context diagram

4.6.2 Internal Interfaces

Internal interfaces are interfaces between the elements that result from the instantiation of design concepts. To identify the relationships and the interface details, you generally need to understand how the elements exchange information at runtime. You can achieve this with the help of modeling tools such as UML sequence diagrams (Figure 4.5), which allow you to model the information that is exchanged

between elements during execution to support use cases or quality attribute scenarios. This type of analysis is also useful for identifying relationships between elements: If two elements need to exchange information directly, then a relationship between these elements must exist. The information that is exchanged becomes part of the specification of the interface. Interfaces typically consist of a set of operations (such as methods) with specified parameters, return values, and possibly, exceptions and pre and post conditions. Some interfaces, however, may involve other information exchange mechanisms, such as a component that writes information to a file or database and another component that then accesses this information. Interfaces may also establish quality of service agreements. For example, the execution of an operation specified in the interface may be time-constrained to satisfy a performance quality attribute scenario.

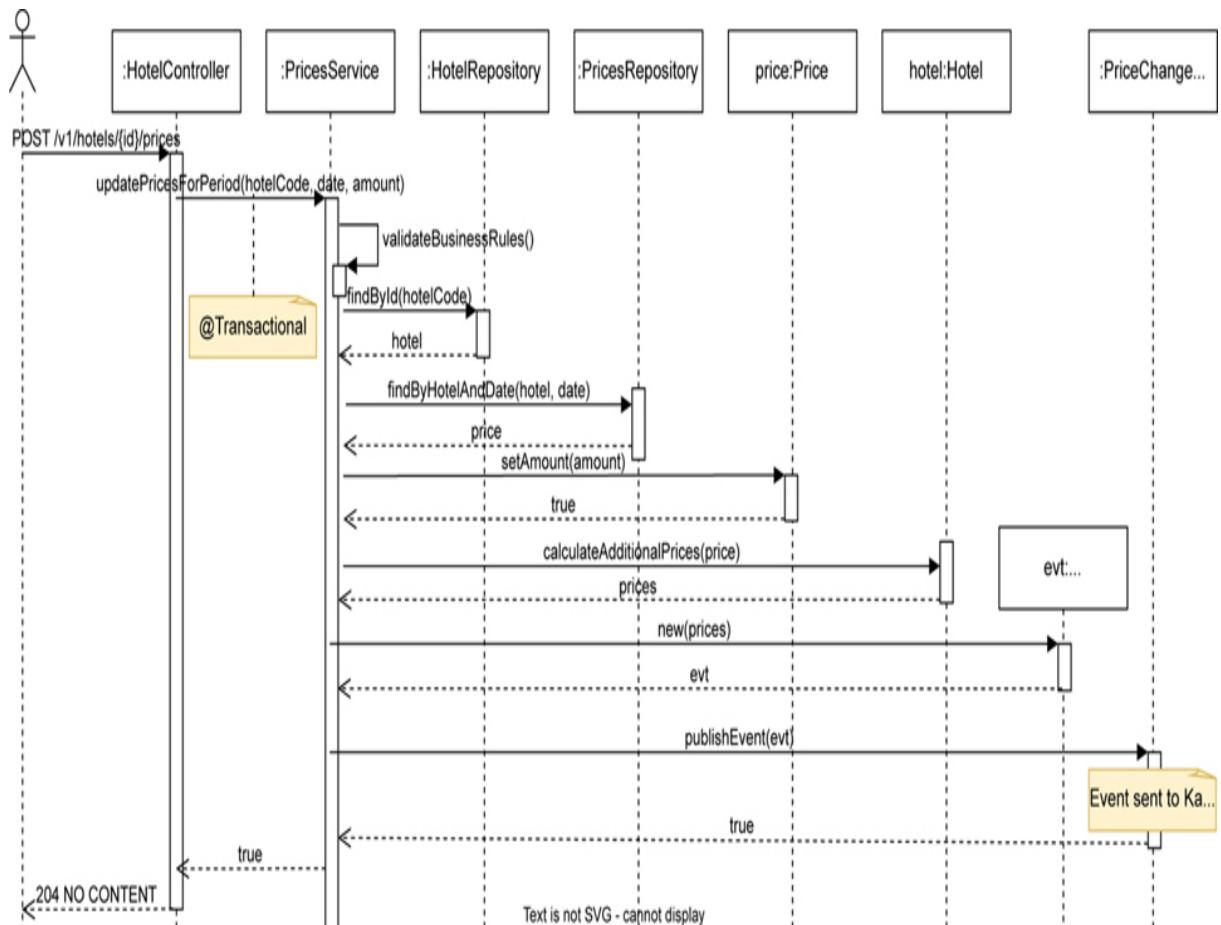


FIGURE 4.5 A sequence diagram used to identify interfaces

The following is an initial sequence diagram for HPS-2 (Change Prices) on the command side. The diagram shows how the client application invokes the POST method that triggers a change in prices. The PriceService retrieves the hotel and price that must be changed and asks the hotel object to calculate additional prices. To increase performance, an index must be added to the database (Increase efficiency of resource usage tactic). Once prices are calculated, an event is created and sent to the PriceChangeEventProcessor which in turn sends it to Kafka.

Key: UML

From this interaction, initial methods for the interfaces of the interacting elements can be identified:

PriceServices:

Method name	Description
	<p>updatePriceForPeriod</p> <p>Update prices for a hotel and produce a price change event.</p> <p>Parameters:</p> <ul style="list-style-type: none">• Hotel Identifier• Range of dates of price changes• Amount <p>Returns:</p> <ul style="list-style-type: none">• true if the update is performed successfully. <p>Throws:</p> <ul style="list-style-type: none">• PriceChangeEventException in case price change fails.

Note: More detail about this example is presented in [Chapter 8](#).

The identification of interfaces is usually not performed equally across all design iterations. When you are starting

the design of a greenfield system, for example, your first iterations will produce only abstract elements such as layers, with these elements then being refined in later iterations. The interfaces of abstract elements such as layers are typically underspecified. For example, in an early iteration you might simply specify that the UI layer sends “commands” to the business logic layer, with the business logic layer sending “results” back. As you advance in the design process and particularly when you create structures to address specific use cases and quality attribute scenarios, you will need to refine the interfaces of the specific elements that participate in the interaction.

In some special cases, identification of interfaces is greatly simplified. For example, when using a persistence framework that provides modules in the data layer, interfaces are already defined by the technology that is selected. The specification of interfaces then becomes a relatively trivial task, as the chosen technologies are designed to interoperate and hence have already “baked in” many interface assumptions and decisions.

Finally, you need to consider that not all of the internal interfaces of the system element will be identified as part of the design process (see the sidebar “[Identifying Interfaces in Element Interaction Design](#)”).

Identifying Interfaces in Element Interaction Design

Although defining interfaces is an essential part of the architecture design process, it is important to recognize that not all of the internal interfaces are identified during architectural design. As part of the architecture design process, you typically consider the primary use cases as part of the architectural drivers, and you identify elements

(usually modules) that support this primary functionality along with the other drivers. This process will, however, not uncover *all* of the elements and interfaces for the system that are required to support the entire set of use cases. This lack of specificity is intended: Architecture is about abstraction, so necessarily some information is less important, particularly in the earliest stages of design.

Identifying the modules that support the nonprimary use cases is often necessary for estimation or work-assignment purposes. Identifying their interfaces is also necessary to support the individual development and integration of the modules and to perform unit testing. This identification of modules may be done early in the project life cycle, but it must not be confused with a big design up front (BDUF) approach. This, at most, is a BDUF that, in certain contexts such as early estimation or iteration planning, is hard to avoid.

As an architect, you may identify the set of modules that supports the complete set of use cases for the system or for a particular release of the system, but the identification of the interfaces associated with the modules that support the nonprimary use cases is typically not your responsibility, as it would require a significant amount of your time and does not usually have a major architectural impact. This task, which we call element interaction design (see [Section 2.2.2](#)), is usually performed after architectural design ends but before the development of (most of) the modules begins. Although this task should be performed by other members in the development team, you play a critical role in it, since these interfaces must adhere to the architectural design that you established. You, as the architect, must communicate the architecture to the engineers who are responsible for identifying the interfaces and ensure that

they understand the rationale for the existing design decisions.

A good way to achieve this communication is to use the *active reviews for intermediate design (ARID)* method. In this method, the architecture design (or part of it) is presented to a group of reviewers—in this case, the engineers who will make use of this design. After the design presentation, a set of scenarios is selected by the participants. The selected scenarios are used for the core of the exercise, where the reviewers use the elements present in the architecture to satisfy them. In standard ARID, the reviewers are asked to write code or pseudo-code for the purpose of identifying interfaces. Alternatively, the architect can present the architecture, select a nonprimary functional scenario and ask the participants to identify the interfaces of the components that support the scenario using sequence diagrams or a similar method.

Aside from the fact that the architectural design is reviewed in this exercise, there are additional benefits to this approach. Specifically, in a single meeting, the architecture design or part of it is presented to the entire team, and agreements can be reached with respect to how the interfaces should be defined (e.g., the level of detail or aspects such as parameter passing, data types, or exception management).

4.7 Creating Preliminary Documentation During Design

A software architecture is typically documented as a set of *views*, which represent the different structures that compose the architecture. The formal documentation of these views is not part of the design process. Structures, however, are produced as part of design. Capturing them,

even in an informal manner (i.e., as sketches), along with the design decisions that led you to create these structures, is a task that should be performed as part of normal design activities.

4.7.1 Recording Sketches of the Views

When you produce structures by instantiating the design concepts that you have selected to address a particular design problem, you will typically not produce these structures in your mind, but rather will create some *sketches* of them. In the simplest case, you will produce these sketches on a whiteboard, a flip-chart, or even a piece of paper. Otherwise, you may use a modeling tool in which you will draw the structures. The sketches that you produce are the initial documentation for your architecture that you should capture and may flesh out later, if necessary. When you create sketches, you don't need to always use a more formal language such as UML. If you use some informal notation, you should at least be careful in maintaining consistency in the use of symbols. Eventually, you will need to add a legend to your diagrams to provide clarity and avoid ambiguity.

You should develop discipline in writing down the responsibilities that you allocate to the elements as you create the structures. The reasons for this are simple: As you identify an element, you are determining some responsibilities for that element in your mind. Writing it down *at that moment* ensures that you won't have to remember it later. Also, it is easier to write down the responsibilities associated with your elements gradually, rather than compiling all of them at a later time.

Creating this preliminary documentation as you design requires some discipline. But the benefits are worth the effort—you will be able to produce the more detailed architecture documentation relatively easily and quickly at a later point. One simple way that you can document responsibilities if you are using a whiteboard, a flip-chart, or a PowerPoint slide is to take a photo of the sketch that you have produced and paste it in a document, along with a table that summarizes the responsibilities of every element depicted in the diagram ([Figure 4.6](#) provides an example). If you are using a computer-aided software engineering (CASE) tool, you can select an element to create and use the text area that usually appears in the properties sheet of the element to document its responsibilities, and then generate the documentation automatically.

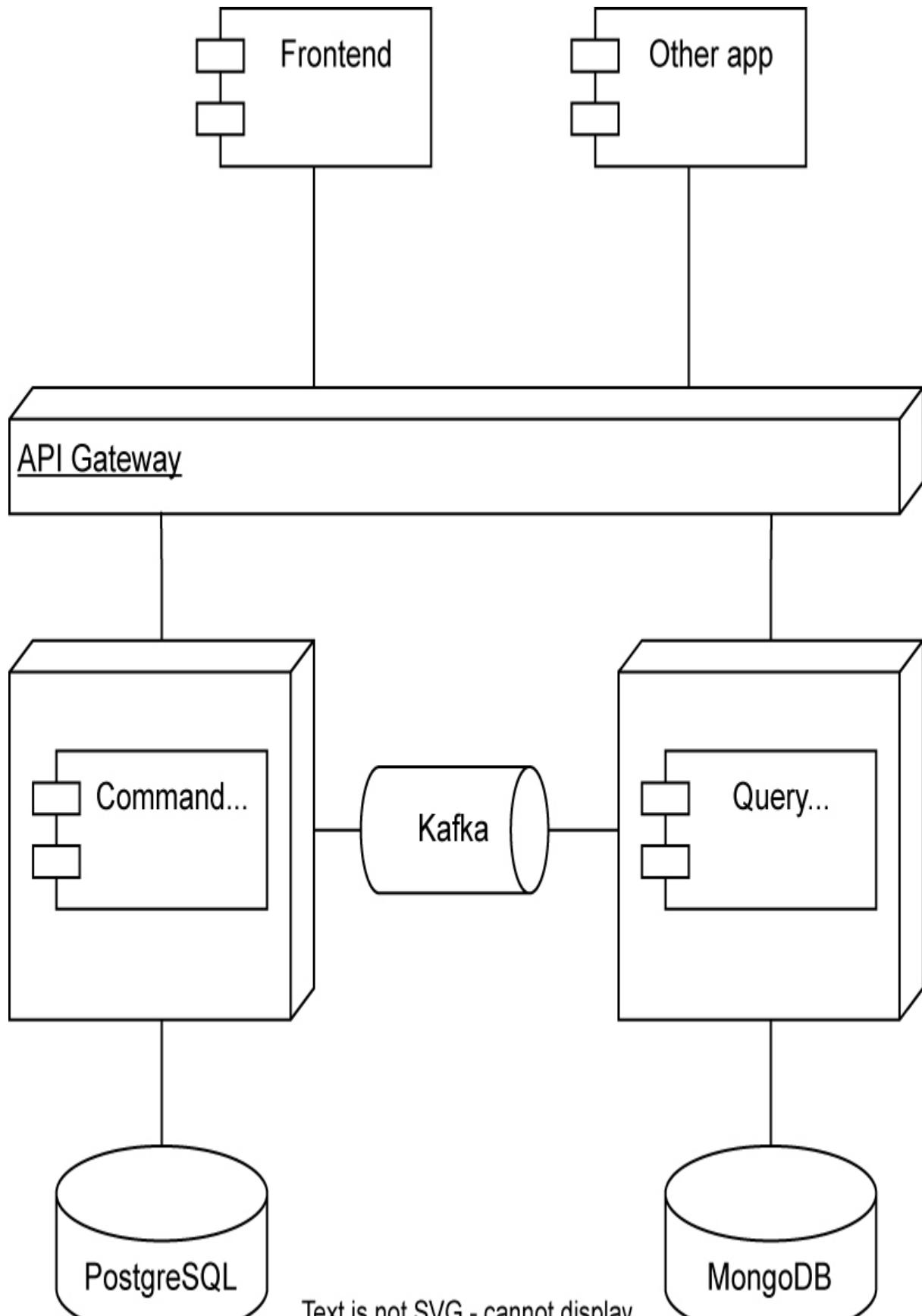




FIGURE 4.6 Sample preliminary documentation

This diagram presents a sketch of a module view depicting the overall system structure from a CQRS-based system. The diagram is complemented with a table that describes the responsibilities of the various elements:

Key: UML

Element	Responsibility
Command Microservice	This microservice is responsible for making changes to the domain entities and sending an event when changes are made .
Query Microservice	This microservice receives events sent by the command microservice and persists the information in the database to support querying.
...	...

Of course, it is not necessary to document *everything*. The three purposes of documentation are analysis, construction, and education. At the moment you are designing, you should choose a documentation purpose and then document to fulfill that purpose, based on your risk mitigation concerns. For example, if you have a critical quality attribute scenario that your architecture design needs to satisfy, and if you will need to prove this requirement is met in an analysis, then you must take care to document the information that is relevant for the analysis to be

satisfactory. Alternatively, if you anticipate having to train new team members, then you should make a sketch of a C&C view of the system, showing how it operates and how the elements interact at runtime, and perhaps construct a crude module view of the system, showing at least the major layers or subsystems.

Finally, it is a good idea to remember, as you are documenting, that your design may eventually be analyzed. Consequently, you need to think about which information should be documented to support this analysis (see the sidebar “[Scenario-Based Documentation](#)”).

Scenario-Based Documentation

An analysis of an architecture design is based on your most important use cases and quality attribute scenarios. Simply put, a scenario is selected and you must explain to reviewers how the architecture supports the scenario, and justify your decisions. To start preparing for the analysis *while* you design, it is useful to produce and document structures that contain the elements that are involved in the satisfaction of a scenario. This should come naturally given that the design process is guided by scenarios, but keeping this point firmly in mind is always helpful.

During the design process, you should at least try to capture the following elements in a single document:

- The primary presentation: the diagram that represents the structure that you produced
- The element responsibilities table: it will help you record the responsibilities of the elements that are present in the structure
- The relevant design decisions, and their rationales (see [Section 4.7.2](#))

You might also capture two other pieces of information:

- A runtime representation of the element's interaction—for example, a sequence diagram
- The initial interface specifications (which can also be captured in a separate document)

As you can see, all of this information needs to be produced as part of the design process. One way or another, you need to decide which elements are present in the system and how they interact. The only question is whether you bother to write this information down, or whether its sole representation is in the code.

If you follow the approach that we advocate here, at the end of the design you will have a set of preliminary views documented, in which each of the views is associated with a particular scenario, and you will have this documentation at little cost. This preliminary documentation can be used “as is” to analyze the design, and particularly through scenario-based evaluations.

4.7.2 Recording Design Decisions

In each design iteration, you make important design decisions to achieve your iteration goal. As we saw previously, these design decisions include the following:

- Selecting a design concept from several alternatives
- Creating structures by instantiating the selected design concept
- Establishing relationships between elements and defining interfaces
- Allocating resources (e.g., people, hardware, computation)

- Others

When you study a diagram that represents an architecture, you see the end product of a thought process, but it may not be easy to understand the decisions that were made to achieve this result. Recording design decisions *beyond* the representation of the chosen elements, relationships, and properties is fundamental to help in understanding how you arrived at the result: the design rationale.

When your iteration goal involves satisfying a specific quality attribute scenario, some of the decisions that you make will play significant roles in your ability to achieve the scenario response measure. These are, therefore, the decisions that you should take the greatest care in recording. You should record these decisions because they are essential to facilitate analysis of the design you created; then to facilitate implementation; and, still later, to aid in understanding of the architecture (e.g., during maintenance). Also every design decision is “good enough” but seldom optimal, so you need to justify the decisions made, and possibly revisit the remaining risks later.

You might think that recording design decisions is a tedious task. In reality, depending on the criticality of the system being developed, you can adjust the amount of information that is recorded. For example, to record a minimum of information, you can use a simple table such as the one shown in [Table 4.2](#). If you decide to record more than this minimum, the following information can prove useful:

- What evidence was produced to justify decisions?
- Who did what?
- Why were shortcuts taken?
- Why were tradeoffs made?
- What assumptions did you make?

Table 4.2 *Example of a Table to Document Design Decisions*

Design Decisions and Location	Rationale and Assumptions
<p>Use a relational database for the Command microservice</p>	<p>Given that the command microservice manages the different entities of the domain model (Hotels, Rates, Room Types, etc...) and given that these entities are related to each other, the most natural and appropriate mechanism for storing them is a relational database.</p> <p>Discarded alternatives are non-relational databases or other types of storage.</p>
<p>Use a non relational database for the Query Microservice</p>	<p>Since the database on the query side does not need to handle transactions and relationships between entities and since the prices that are stored have a variable structure depending on the hotel, it was decided that the use of a non-relational database is more appropriate. PriceChangeEvents can be appropriately stored in a document database.</p> <p>Discarded alternatives include other types of NoSQL, relational databases or other types of storage.</p>

And, in the same way that we suggest you record responsibilities as you identify elements, you should record the design decisions as you make them. The reason for this is simple: If you leave it until later, you may not remember why you did things.

4.8 Tracking Design Progress

Even though ADD provides clear guidelines to perform design systematically, it does not provide a mechanism to track design progress. When you are performing design, however, there are several questions that you want to answer:

- How much design do we need to do?
- How much design has been done so far?
- Are we finished?

Agile practices such as the use of backlogs and Kanban boards can help you track the design progress and answer these questions. These techniques are not limited to Agile methods, of course. Any development project using any methodology should track progress.

4.8.1 Use of an Architectural Backlog

The concept of an architecture (or design) backlog has been proposed by several authors. This is similar to the product backlog that is found in Agile development methods such as Scrum, but the focus here is not on feature development. The basic idea is that you need to create a list of the pending actions that still need to be performed as part of

the architecture design process. These actions are called “enablers” in methods such as the Scaled Agile Framework.

Initially, you should populate the design backlog with tasks focused on directly satisfying drivers, but other activities that support the design of the architecture can also be included. For example:

- Creation of a prototype to test a particular technology or to address a specific quality attribute risk
- Exploration and understanding of existing assets (possibly requiring reverse engineering)
- Issues uncovered in a review of the design
- Review of a partial design that was performed on a previous iteration

For example, when using Scrum, the sprint backlog and the design backlog are not independent: Some features in the sprint backlog may require architecture design to be performed, so they will generate items that go into the architectural design backlog. While these two backlogs can be managed separately, it may be more practical to combine the architectural tasks with features and other elements in the backlog (see figure “Value and Visibility: The Dimensions of Entries in Your Project Backlog” in [section 10.2](#)).

Also, additional architectural concerns may arise as decisions are made. For example, if you choose a reference architecture, you will probably need to add specific architectural concerns, or quality attribute scenarios derived from them, to the architectural design backlog. An example of such a concern is the establishment of a mechanism to support observability.

4.8.2 Use of a Design Kanban Board

As design is performed in rounds and as a series of iterations within these rounds, you need to have a way of tracking the design's degree of advancement. You must also decide whether you need to continue making more design decisions (i.e., performing additional iterations). One tool that can be used to facilitate this task is a Kanban board, such as the one shown in [Figure 4.7](#).



FIGURE 4.7 An example of a Kanban board used to track design progress

At the beginning of a design round, the inputs to the design process become entries in the backlog. Initially, that activity

occurs in step 1 of ADD; the different entries in your backlog for this design round should be added to the “Not Yet Addressed” column of the board (except if you have some entries that were not concluded in previous design rounds that you wish to address here). When you begin a design iteration, in step 2 of ADD, the backlog entries that correspond to the drivers that you plan to address as part of the design iteration goal should be moved to the “Partially Addressed” column. Finally, once you finish an iteration and the analysis of your design decisions reveals that a particular driver has been addressed (step 7 of ADD), the entry should be moved to the “Completely Addressed” column of the board. It is important to establish clear criteria that will allow a driver to be moved to the “Completely Addressed” column (think of this as the “Definition of Addressed” criteria, similar to the “Definition of Done” criteria used in Scrum). A criterion may be, for example, that the driver has been analyzed or that it has been implemented in a prototype. Also, drivers that are selected for a particular iteration may not be completely addressed in that particular iteration, in which case they should remain in the “Partially Addressed” column and, in preparation for subsequent iterations, you should consider how they can be allocated to the elements that exist at this point.

It can be useful to select a technique that will allow you to differentiate the entries in the board according to their priority. For example, you might use different colors of Post-it notes depending on the priority.

With such a board, it is easy to visually track the advancement of design, as you can quickly see how many of the (most important) drivers are being or have been addressed in the design round. This technique also helps you decide whether you need to perform additional

iterations as, ideally, the design round is terminated when a majority of your drivers (or at least the ones with the highest priority) are located under the “Completely Addressed” column.

4.9 Summary

In this chapter, we presented a detailed walk-through of the Attribute-Driven Design method, version 3.0. We also discussed several important aspects that need to be considered in the various steps of the design process. These aspects include the use of a backlog, the various ways that ADD can be applied to different system contexts, the identification and selection of design concepts and their use in producing structures, the definition of interfaces, and the production of preliminary documentation.

Even though the overall architecture development life cycle includes documenting and analyzing architecture as activities that are separate from design, we have argued that a clean separation of these activities is artificial and harmful. We stress that preliminary documentation and analysis activities need to be regularly performed as integral parts of the design process.

4.10 Further Reading

Some of the concepts of ADD 3.0 were first introduced in: H. Cervantes, P. Velasco, and R. Kazman, “A Principled Way of Using Frameworks in Architectural Design,” *IEEE Software*, 46–53, March/April 2013.

Version 2.0 of ADD was first documented in the SEI Technical Report: R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord, and B. Wood, “Attribute-Driven Design (ADD), Version 2.0,” SEI/CMU Technical

Report CMU/SEI-2006-TR-023, 2006. An extended example of using ADD 2.0 was documented in W. Wood, "A Practical Example of Applying Attribute-Driven Design (ADD), Version 2.0," SEI/CMU Technical Report: CMU/SEI-2007-TR-005.

The concept of an architecture backlog is discussed in C. Hofmeister, P. Kruchten, R. Nord, H. Obbink, A. Ran, and P. America, "A General Model of Software Architecture Design Derived from Five Industrial Approaches," *Journal of Systems and Software*, 80:106–126, 2007.

The ARID method is discussed in P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2002.

The CBAM method is presented in L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed., Addison-Wesley, 2013.

The ways in which an architecture can be documented are covered extensively in P. Clements et al. *Documenting Software Architectures: Views and Beyond*, 2nd ed., Addison-Wesley, 2011. More Agile approaches to documenting are discussed in books such as S. Brown, *Software Architecture for Developers*. Lean Publishing, 2015.

The importance and challenges of capturing design rationale are discussed in A. Tang, M. Ali Babar, I. Gorton, and J. Han, "A Survey of Architecture Design Rationale," *Journal of Systems and Software*, 79(12):1792–1804, 2007. A minimalistic technique for capturing rationale is discussed in U. Zdun, R. Capilla, H. Tran, and O. Zimmermann, "Sustainable Architectural Design Decisions," *IEEE Software*, 30(6):46–53, 2013.

4.11 Discussion Questions

1. In step 2 of ADD; what can happen if the goal that is selected for the iteration involves a scope that is too big?
2. In the design of greenfield systems for mature domains ([section 4.3.1](#)), the recommendation is to first identify structures to support primary functionality before addressing other drivers, including quality attributes. Why is this recommendation made? Could you design to support quality attributes before considering how primary functionality will be supported?
3. What are the consequences of the architect deciding that he or she wants to perform ADD iterations for *all* of the system's use cases?
4. What are the risks of not performing step 6 of ADD and leaving all documentation activities for later?
5. What are the benefits or downsides of using an architectural backlog and its associated Kanban board versus just adding architectural tasks to the standard product backlog used in Scrum?

5. Supporting Business Agility through API-Centric Design [This content is currently in development.]

This content is currently in development.

6. Designing for Deployability

In [Chapter 5](#) we focused on the importance of composability and how API-centric design supports business agility. But just designing your infrastructure and systems with APIs and separation of concerns in mind may not be enough. A focus on composability, if done well, allows you to change your code base with relative ease, and can help to avoid accumulating technical debt. But composability, by itself, doesn't get those changes out to market. For that we need deployability. In this chapter, we describe deployability as a quality attribute, present some design concepts associated with it and discuss it in the context of ADD.

Why read this chapter?

Today, DevOps practices have become widespread and are key to many business models, shortening the development life cycle. Deployability is an essential quality attribute related to these practices. In this chapter you'll become familiar with this quality attribute and how to design to achieve it at scale using ADD.

6.1 Deployability Principles and Architectural Design

Deployability, like any other important system quality, needs to be consciously designed and managed. How do we do this? First let's understand what we mean by this term.

6.1.1 Defining deployability

Deployability refers to a property of software indicating that it may be deployed—allocated to an environment for execution—with predictable time and effort. If the deployment process is fully automated—that is, if there is no human intervention—then it is called *continuous deployment*. An architect who is designing for deployability needs to reason about the deployment pipeline: the sequence of tools and activities that begin when a developer checks in some code and ends when some application functionality has been deployed to an execution environment. Deployability is not simply a quality attribute that is associated with products, but also with the components that are used in those products (and which might implement the APIs discussed previously).

Deployment means more than just delivering the system or its components to the *final* environment. As part of the development process, the system needs to be deployed to different environments in addition to the one where the system is developed. Different sets of tests are performed in each environment, expanding the testing scope from unit testing of a single module in the development environment, to functional testing of all the components that make up a service in the integration environment, and ending with broad quality testing in the staging environment and usage monitoring in the production environment. The major

environments that typically exist and the tests that are performed in each one of them are as follows (see [figure 6.1](#)):

- Code is written in a *development environment* for a module where it is subject to unit tests. After it passes the tests, and potentially after appropriate review, the code is pushed to a version control system that triggers the build activities in the integration environment.
- An *integration environment* is a pre-production environment which builds an executable version of the component or service. A continuous integration server compiles the changed code, along with the latest compatible versions of code for other portions of the service and constructs a deployable executable image. Tests in the integration environment may include the unit tests for the various modules (now run against the built system), as well as integration tests designed specifically for the entire system or system part being deployed. When the tests are passed, the built service is promoted to the staging environment.
- A *staging environment* is a pre-production environment which tests for various qualities of the total system. These include performance testing, security testing, license conformance checks, and possibly user testing. A system that passes all staging environment tests—which may include field testing—is deployed to the production environment, typically using either a blue/green model or a rolling upgrade. In some cases, partial deployments are used for quality control or to test the market response to a proposed change or offering.
- Once in the *production environment*, the service is monitored until all parties have confidence in its quality.

At that point, it is considered a normal part of the system.

These environments, along with the tools to provision them, to move artifacts from one environment to another, to test them, and to monitor them, form the backbone of the deployment pipeline.

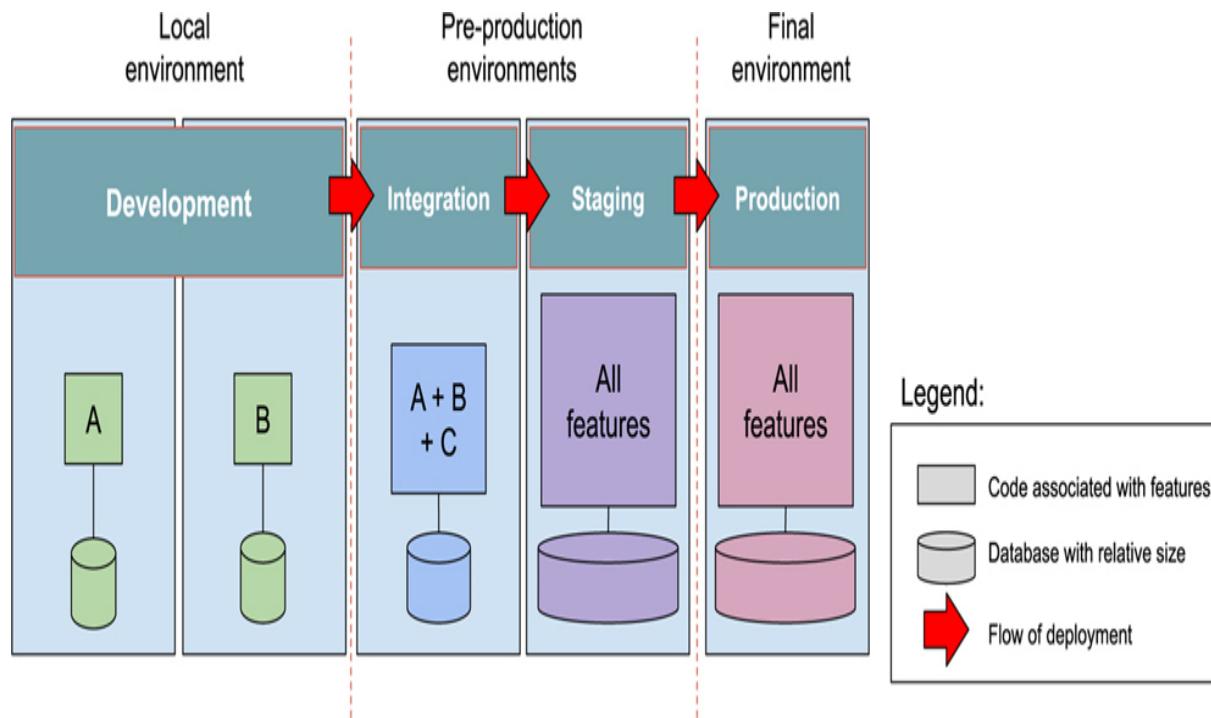


FIGURE 6.1 Environments where the system is deployed

The general scenario for Deployability is shown in [Table 6.1](#), and part of this scenario involves specifying which environment is of concern. But, of course, other concerns need to be reflected as well: what causes the trigger for a deployment, what needs to be changed, what should happen in response to the trigger, and how we measure success.

Table 6.1 General Scenario for Deployability

Portion of Scenario	Description	Possible Values
Source	The trigger for the deployment	End user, developer, system administrator, operations personnel, component marketplace, product owner.
Stimulus	What causes the trigger.	A new element is available to be deployed. This is typically a request to replace a software element with a new version, e.g., fix a defect, apply a security patch, upgrade to the latest release of a component or framework, upgrade to the latest version of an internally-produced element. New element is approved for incorporation. An existing element/set of elements needs to be rolled back.
Artifacts	What is to be changed	Specific components or modules, the system's platform, its user interface, its environment, or another system with which it interoperates. Thus the artifact might be a single software element, multiple software elements, or the entire system.
Environment	Where the artifacts are being deployed	Integration environment Staging environment Production environment.
Response	What should happen	Full deployment. Subset deployed to a specified portion of: users, VMs, containers, servers, platforms. Monitor the new components Roll back a previous deployment.
Response Measure	How effective is the deployment, in terms of	Cost in terms of: <ul style="list-style-type: none">• number, size, complexity of affected artifacts

cost, time, or process effectiveness	<ul style="list-style-type: none"> • average/worst-case effort • elapsed time • money (direct outlay or opportunity cost) • new defects introduced <p>Extent to which this deployment/rollback affects other functions or quality attributes</p> <p>Number of failed deployments</p> <p>Repeatability of the process</p> <p>Traceability of the process</p>
--------------------------------------	---

Note that while we refer to the quality attribute as deployability, there is a set of practices that have grown up around the concerns of deployability that have become known as DevOps, because they bridge the concerns of Development and Operations. DevOps includes additional concerns such as monitoring the system in production.

6.1.2 Continuous integration, deployment and delivery

Frequently delivering new versions of the system, or its components, is necessary to support agile development. This means that a change that is made in the development environment needs to be deployed as quickly as possible to the production environment and, consequently, it needs also to move across the different pre-production environments discussed previously. This is a principle of DevOps and a key aspect to support it is to *automate* this process as much as possible.

When the process is fully automated—that is, if there is no human intervention—then it is called *continuous integration (CI)* or *continuous deployment (CD)*, depending on how far automation takes the system across the environments. If the process is automated up to the point of placing system components into production and human intervention is required (perhaps due to regulations or policies) for this final step, then the process is called *continuous delivery*.

In many systems, deliveries can occur at any time—possibly multiple ones per day—and each can be instigated by a different team. Being able to deliver frequently means that new features, bug fixes, and security patches do not have to wait until the next scheduled release, but can be made and released as soon as the feature is tested and ready, or as a bug is discovered and fixed.

Continuous deployment is not desirable, or even possible, in all domains. If the software exists in a complex ecosystem with many dependencies, it may not be possible to release just one part of it without coordinating that release with the other parts. In addition, many embedded systems, safety-critical systems, systems residing in hard-to-access locations, and systems that are not networked would be poor candidates for a continuous deployment mindset. In such cases a strategy of frequent deployment to staging, with less frequent deployments to production could be employed.

If the new deployment is not meeting its specifications, it can be (and in some cases must be) rolled back. This “rolling back” would return the system to its prior state, within a predictable and acceptable amount of time and effort. As the world moves increasingly toward virtualization and cloud infrastructures (see [Chapter 7](#)), and as the scale of deployed software-intensive systems increases, it is one of the architect’s responsibilities to make design decisions

to support efficient and predictable deployment. In particular, the ability to roll back deployments quickly and efficiently is key to minimizing overall system risk.

6.1.3 Designing for deployability

To design for deployability decisions need to be made about both: 1) the architecture of the system being deployed, and 2) the architecture of the deployment infrastructure. While these architectures are related, they are distinct and may be separately designed. We will discuss the deployment infrastructure in [section 6.1.3.2](#).

6.1.3.1 Designing the system to support deployability

To support deployability, an architect needs to make decisions about how an executable is updated on a host platform, as well as how it is subsequently invoked, measured, monitored, and controlled. Mobile systems in particular present a challenge for deployability in terms of how they are updated because of bandwidth and data volume constraints. Some of the issues involved in deploying software are as follows:

- How does the new executable arrive at its host (e.g., push, where updates deployed are unbidden, or pull, where users or administrators must explicitly request updates)?
- How does it interact with any existing systems? Can this interaction be switched over while the existing systems are executing?
- What is the medium, such as USB drive or Internet delivery?

- What is the packaging (e.g., executable, container image, app, plug-in)?
- What is the resulting integration into an existing system?
- What is the efficiency of executing the process?
- How controllable and automated is the process? For example, can new versions be swapped in and out, and potentially rolled back, within minimal human intervention, without disrupting system operations?

With all of these concerns, the architect must be able to assess the associated risks. Architects are primarily concerned with the degree to which the architecture supports deployments that are:

- *Granular*—Deployments can be of the whole system or of elements within a system. If the architecture provides options for finer granularity of deployment, then certain risks can be reduced.
- *Controllable*—The architecture should provide the capability to deploy at varying levels of granularity, monitor the operation of the deployed units, and roll back unsuccessful deployments.
- *Efficient*—The architecture should support rapid deployment (and, if needed, rollback) with a reasonable level of effort.

Architectural choices (tactics and patterns) profoundly affect deployability. For example, by employing the microservice architecture pattern (see [6.2.2](#)), each team responsible for a microservice can package an executable with all its runtime dependencies and hence can make their own technology choices. This removes incompatibility problems that would previously have been discovered at integration time (e.g., incompatible choices of which version of a library to use).

Since microservices are independent services, such choices do not normally cause problems.

Similarly, a continuous deployment mindset forces you to think about the infrastructure for testing, and you are wise to think about this early in the development process. This “shift left” is necessary because designing for continuous deployment requires continuous automated testing. In addition, the need to be able to roll back or disable features leads to architectural decisions about mechanisms, such as feature toggles and backward compatibility of interfaces. These decisions are best taken early on.

6.1.3.2 Designing the deployment infrastructure

Deployability requires the support of a deployment infrastructure which is primarily constituted by a *deployment pipeline*. A deployment pipeline is the sequence of tools and activities that begin when you check your code into a version control system and end when your system has been deployed so that users or other systems can send it requests.

In between those endpoints, a series of tools build, integrate, and automatically test the newly committed code, test the integrated code for functionality, and test the system for concerns such as performance under load, security, and license compliance and, when quality gates are met, push the new services into production.

Furthermore, things do not always go according to plan. If you find problems after the software is in its production environment, perhaps by monitoring or perhaps by user-submitted bug reports, it may be necessary to roll back to a previous version while the defect is being addressed. Ideally we want this rollback to be done at the push of a button,

and so this capability must also be part of the deployment pipeline.

Thus tools to realize a deployment pipeline include code analysis tools, test automation tools, continuous integration servers, deployment automation tools, monitoring tools, and artifact repositories. It should be noted that the design of the deployment infrastructure is primarily a task of DevOps (or DevSecOps) engineers in collaboration with the architects. Of course, in smaller organizations these might be the same person, but in large organizations they will be separate individuals or even separate teams.

6.2 Design Decisions to Support Deployability

To design for deployability we need to think about all of the bottlenecks that might prevent us from deploying quickly and with minimal labor. We want to ensure that our software is:

- Easy to change, with few ripple effects, and easy for developers to work in parallel. This increases delivery velocity as one developer's changes are less likely to affect another, thus testing and deployment are likely to be less complex.
- Easy to test with automated tests. If tests are not fully automated, the pipeline will require human intervention, potentially reducing velocity.
- Easy to monitor, so that defects introduced by the release, such as performance problems or security flaws, can be quickly spotted.
- Easy to roll back if, for example, the monitoring revealed a problem.

- Easy to scale, up or down, so that a gradual replacement of instances can be made without affecting the performance of the running application.
- Robust to failures of individual parts, so that a failed update in a single service will not cause the entire system to fail, for example.

Examining this list, it should come as no surprise that these are all qualities—modifiability, performance, availability, testability—that architects have striven to achieve for decades. These are all desirable for many complex systems. The only aspect of them that is particular to deployability is that we want all of them, all the time! In this chapter we only focus on the QAs that are specific to deployability, but keep in mind that if you do not worry about the other quality dimensions you will likely not achieve your deployability goals.

6.2.1 Deployability Tactics

A deployment may be triggered by the release of one or more new or modified software elements. The deployment is successful if these elements are deployed within acceptable time, cost, and quality constraints.

[Figure 6.2](#) shows the deployability tactics categorization. There are two major categories of deployability tactics: *Manage Deployment Pipeline*, and *Manage Deployed System*. The first category is focused on design decisions that apply to the deployment infrastructure, while the second is focused on design decisions that apply to the functionality being deployed. Let us examine these in turn.

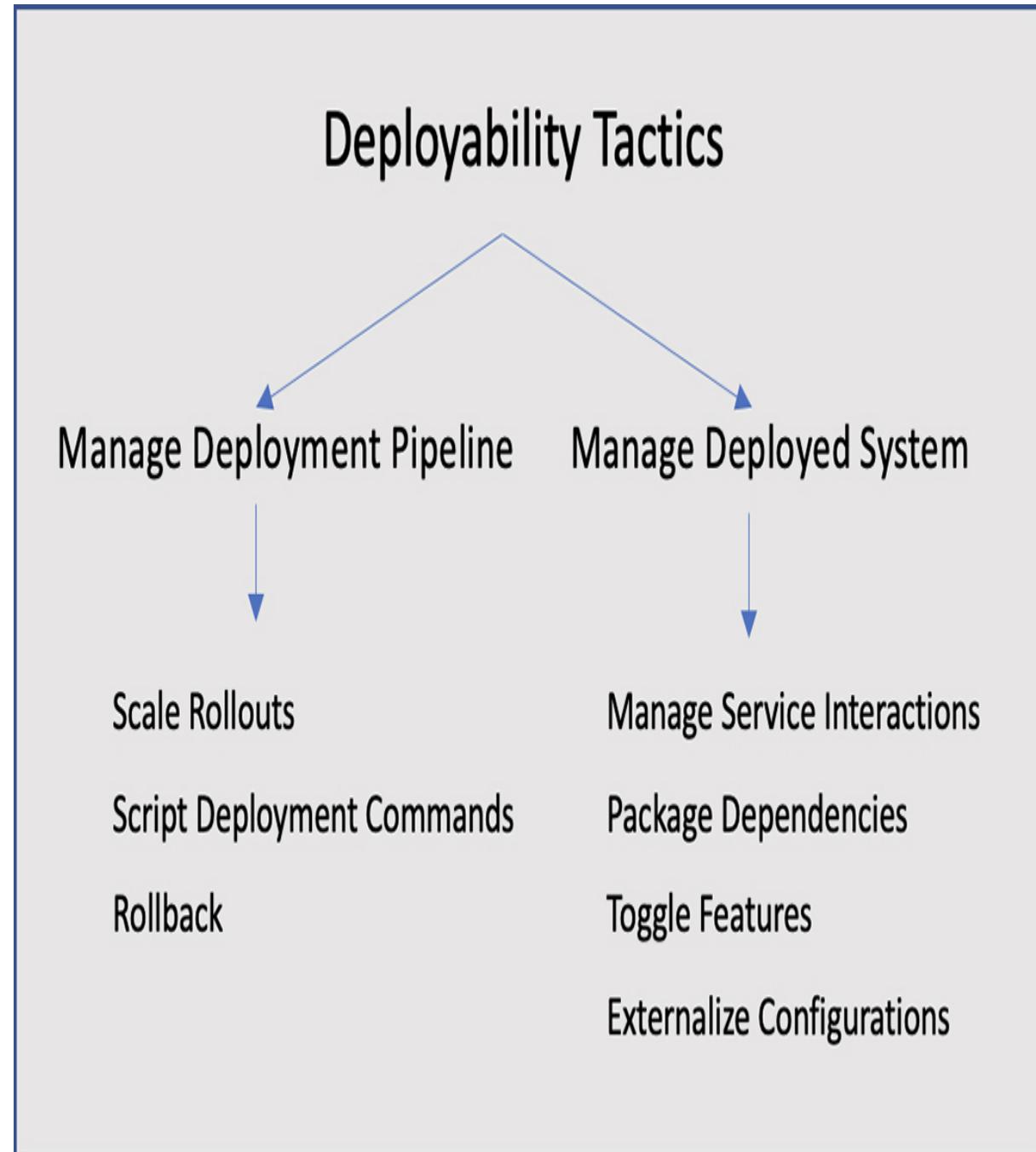


FIGURE 6.2 Deployability Tactics Categorization

Within the Manage Deployment Pipeline category the tactics are:

- *Scale rollouts*. Rather than deploying to the entire user base, scaled rollouts deploy a new version of a service gradually, to subsets of the user base. By gradually

releasing, the effects of new deployments can be monitored and measured and, if necessary, rolled back.

- *Roll back*. If a deployment has defects it can be “rolled back” to its prior state. Since deployments may involve multiple updates of multiple services, the rollback mechanism must be able to keep track of these, or reverse the consequences of any update, ideally in a fully automated fashion.
- *Script deployment commands*. The complex steps to be carried out and precisely orchestrated in a deployment should be scripted. This is part of a general movement towards “Infrastructure as Code”.

Within the Manage Deployed System category the tactics are:

- *Package dependencies*. Elements are packaged with their internal dependencies (those required for the element to execute) so that they get deployed together.
- *Manage service interactions*. Deployment of versions of services that are interacting with external entities needs to be mediated so that incompatibilities are avoided.
- *Feature toggle*. A “kill switch” (feature toggle) can be used for new features to automatically disable a feature at runtime, without forcing a new deployment. It is often the case that issues arise after deploying new features. Feature toggles reduce the risk associated with such bugs.
- *Externalize configurations*. The system should not have any hardcoded configurations that limit the possibility of moving it from one environment to another (e.g. from testing to production). This is typically achieved through the use of environment variables whose values are stored in an external repository.

6.2.2 Deployability Patterns

Architecture patterns for deployability can be organized into two major categories, similar to the two categories of tactics. The first category contains patterns for structuring services to be deployed. The second category contains patterns for how to deploy services, which can be parsed into two broad subcategories: all-or-nothing or partial deployment. The two main categories for deployability are not completely independent of each other because certain deployment patterns depend on certain structural properties of the services.

We begin by discussing patterns for structuring the services to be deployed. With respect to deployability concerns, there are a few common options for structuring a system: as a set of microservices, as a set of tiers, as a load-balanced cluster, or as a (possibly modular) monolith. We briefly sketch each of these architectural patterns.

6.2.2.1 Microservice architecture

The microservice architecture pattern structures the system as a collection of independently deployable services that communicate only via messages through service interfaces. These services are meant to be:

- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around business capabilities
- Owned by a small team

There is no other form of interprocess communication allowed: no direct linking, no direct reads of another

service's data store, no shared-memory model, no back-doors whatsoever. Services are usually stateless, and (because they are developed by a single relatively small team) are relatively small—hence the term microservice. Service dependencies should be acyclic. An integral part of this pattern is a discovery service so that messages can be appropriately routed.

Microservices provide APIs and they may also need to consume APIs to satisfy the contract that they provide. These APIs are generally of request-response type, but they may also be asynchronous (sending and receiving events). Consumed APIs, especially of request-response type, become dependencies of the microservice.

One of the biggest challenges of using microservices is to find the right granularity of the microservices and the APIs they provide. There are a number of drivers for granularity disintegration and integration. Granularity disintegration means that a service should be broken apart while granularity integration means that a service should be integrated from separate services. Granularity disintegration drivers include the following:

- *Service scope and function*: is the service doing unrelated things?
- *Code volatility*: are changes isolated to only one part of the service?
- *Scalability and throughput*: do parts of the service need to scale differently?
- *Security*: do some parts of the service need higher security levels than others?
- *Extensibility*: is the service always expanding to add new contexts?

Granularity integration drivers include the following:

- *Database transactions*: is an ACID transaction required between separate services?
- *Workflow and choreography*: do services need to talk to one another, and are there services that interact highly with each other?
- *Shared code*: do services need to share code between each other?
- *Data relationships*: although a service can be broken apart, can the data it uses be broken apart as well?

Management of concerns that arise because of dependencies between microservices is also a challenging aspect. For example, consumed APIs, both external and internal, may require credentials to be used, so design decisions on how to provide these credentials to the consumed APIs need to be made. There are different token delegation patterns to address this situation. Another challenging concern is error management. If an invocation to a consumed API fails, this may result in an error in the provider of the API. Propagating the error information across dependent microservices so that the error can be tracked adequately, and so that the end user can obtain useful information is also a challenge. Failures in the consumption of an API can be also problematic, and they can be addressed using patterns such as the Circuit Breaker pattern. Transaction management is also a challenging situation if an API endpoint requires the implementing microservice to perform a transaction that involves two or more consumed APIs, these are usually addressed using patterns such as SAGA.

Many other concerns need to be dealt with when using microservices, including service discovery, load balancing, scaling, monitoring, management of configuration among others. One approach to dealing with these concerns is the

introduction of *sidecars* as proxies between dependent microservices. In this way, code that supports these concerns is placed in the sidecar and is not part of the microservice. These sidecars intercept data according to rules defined by a controller; this is the basis of a Service Mesh.

6.2.2.2 Monoliths and modular monoliths

A monolithic architecture is one where all of the modules of the system are compiled into a single executable which is deployed as a single unit. Any code change typically requires a re-deployment of the entire system. This was the way that most systems were structured before the advent of microservice architectures. As with any architectural pattern, monoliths have costs and benefits. In recent years many organizations have moved away from this approach. Some of the reasons for this migration away from monoliths is that they allow for less control over deployment, and fewer options for scalability.

One possible benefit of using a monolith is that, since it is not a distributed system, there should be fewer problems related to distributed transactions that occur frequently with microservice architectures. Also managing the deployment of a single unit of code is simpler than managing the deployment of multiple units of code. The complexity associated with microservice architectures has motivated some organizations to move back to monoliths in recent years.

One variant of traditional monoliths which has become popular recently is Modular Monoliths. In modular monoliths, modules contain all needed parts to provide particular features, and the data associated with the module is isolated from the data belonging to other modules, similar to how functionality and data are packaged together in

microservices. Modular monoliths are thus similar to microservices, with the difference that the modules are not deployed as independent executables, and they do not result in a distributed system, also, the data they store is usually isolated (see [Figure 6.3](#)). While modular monoliths still suffer the same limitations as traditional monoliths in terms of deployment and scalability, one benefit of this approach is that the internal modularization of these monoliths should facilitate their separation into different deployment units in case this is required later on.

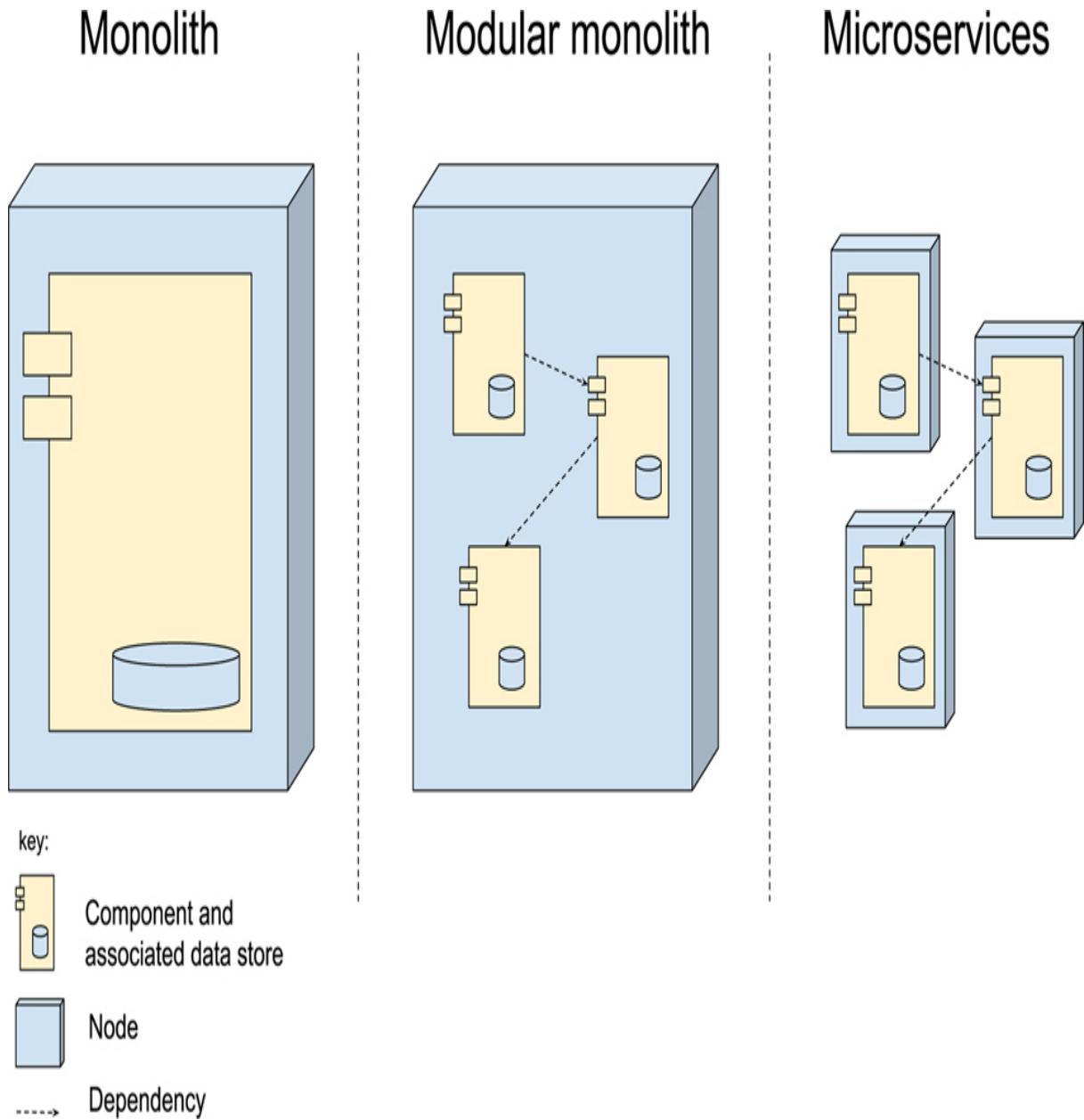


FIGURE 6.3 Monolith vs Modular Monolith vs Microservices.

Now we turn our attention to a different class of patterns. These are patterns for managing the deployment of services. The first two patterns presented above, microservices and monoliths, are options for how to structure a system or a service. The patterns below are about how to deploy a system or service, irrespective of how it is internally structured.

Suppose there are N instances of Service A and you wish to replace them with N instances of a new version of Service A, leaving no instances of the original version. You wish to do this with no reduction in quality of service to the clients of the service, so there must always be N instances of the service running.

6.2.2.3 N-Tier Deployment

Another common form of deployment, often chosen to facilitate performance scalability, is n-tier deployment. The creation of tiers involves additional costs, in terms of added network latency, complexity, and deployment effort. But it offers many benefits, such as the ability to optimize and scale different tiers independently, to isolate faults, and to promote security, by placing sensitive resources in inner tiers that are not directly accessible by external clients. In addition, different security policies may be applied according to the tier and firewalls may be added between the tiers. Three common alternatives of distributed deployment are 2-tier, 3-tier, and 4-tier.

2-Tier deployment (or Client-server)

This is the most basic layout for distributed deployment. The client and the server are usually deployed on different physical tiers, as shown in [figure 6.4](#).

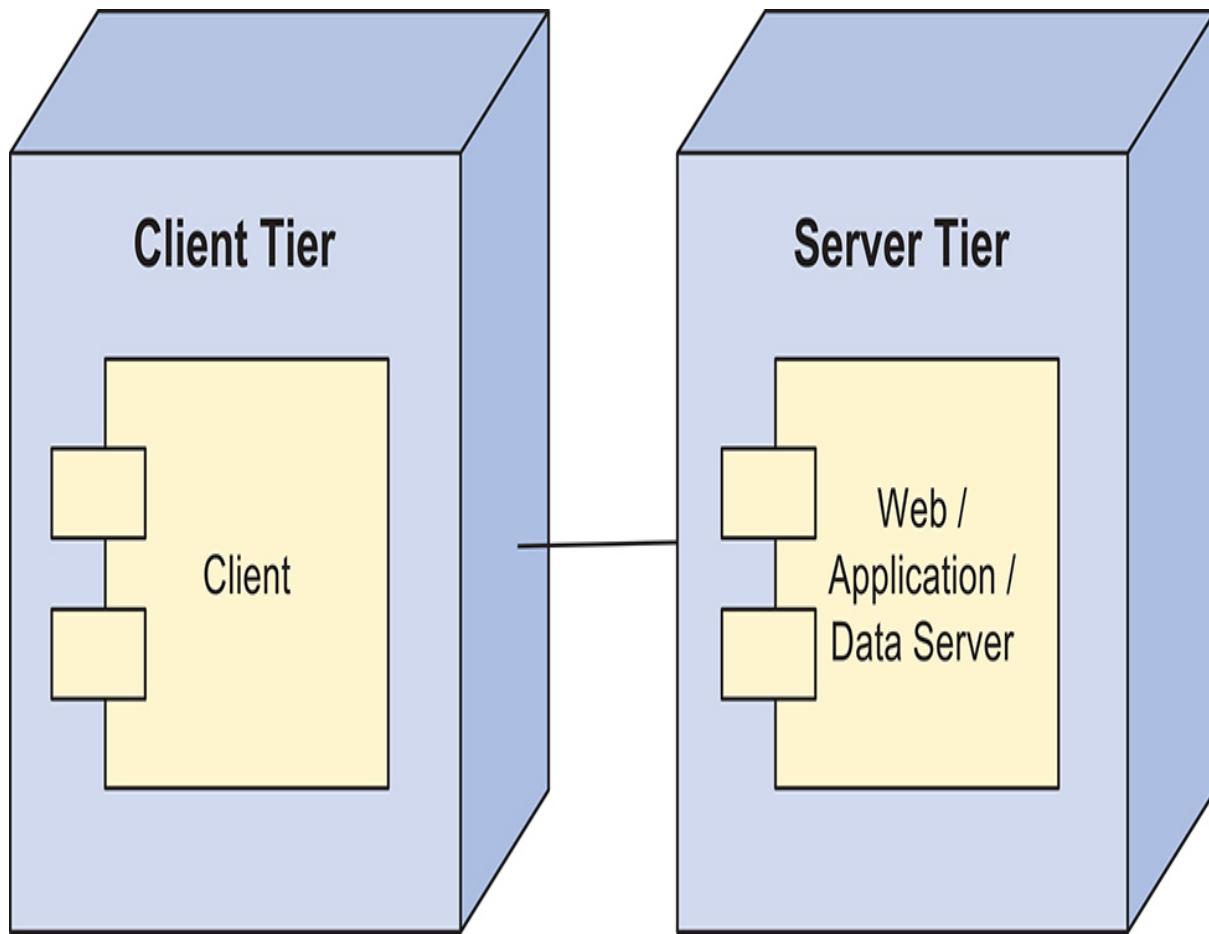


FIGURE 6.4 An Example 2-Tier Deployment.

3-Tier Deployment

In this pattern, shown in [figure 6.5](#), the application is deployed in a tier that is separate from the one that hosts the database. This is a very common physical layout for web applications.

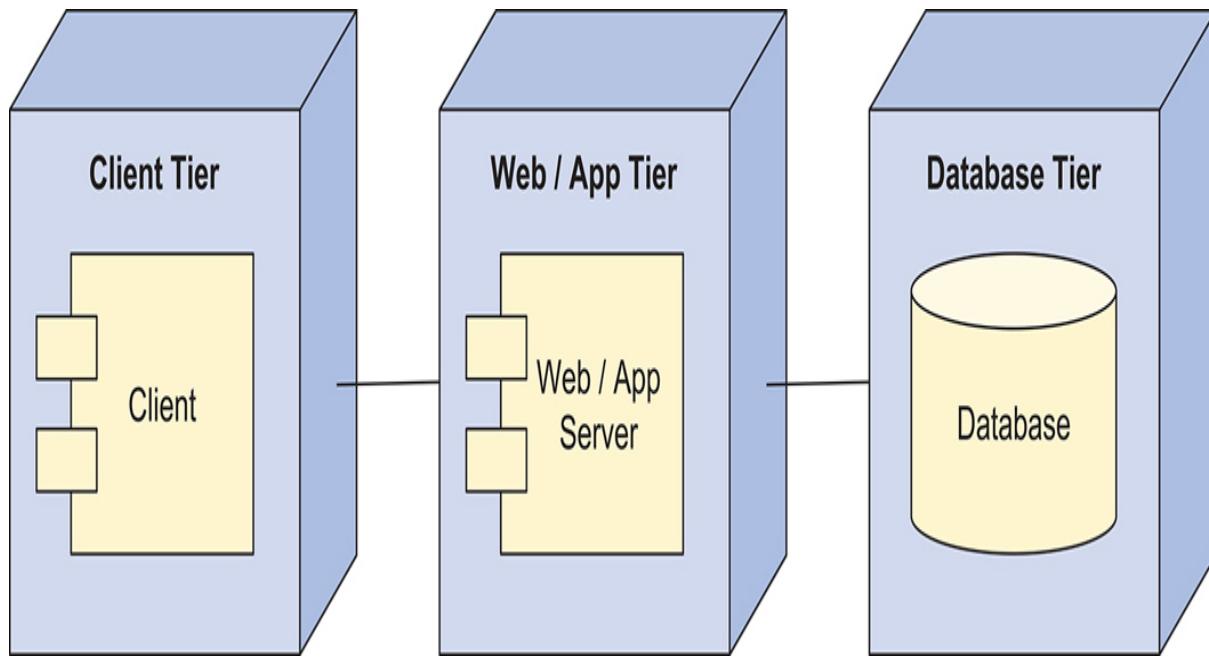


FIGURE 6.5 An Example 3-Tier Deployment.

4-Tier Deployment

In this variant, shown in [figure 6.6](#), the Web server and the application server are deployed in different tiers. This separation is usually done to improve security as the web server may reside in a publicly accessible network while the application resides in a protected network. Additionally, firewalls may be placed between the tiers.

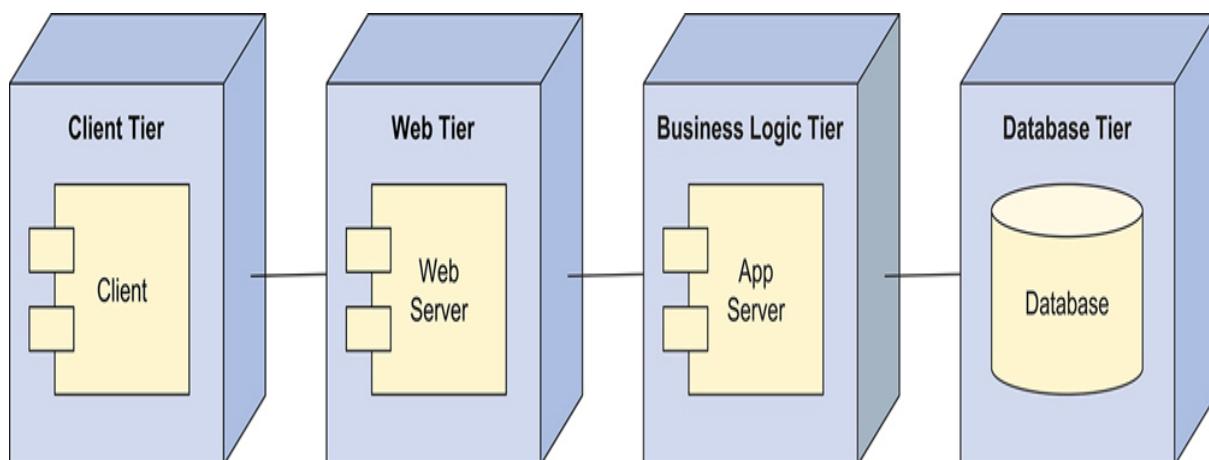


FIGURE 6.6 An Example 4-Tier Deployment.

6.2.2.4 Load-balanced Cluster

In this deployment pattern, an application is deployed onto multiple servers that share the workload (see [figure 6.7](#)). Client requests are received by a load balancer which redirects them to one of the servers, according to their current load. The application server instances can process requests concurrently which results in performance improvements.

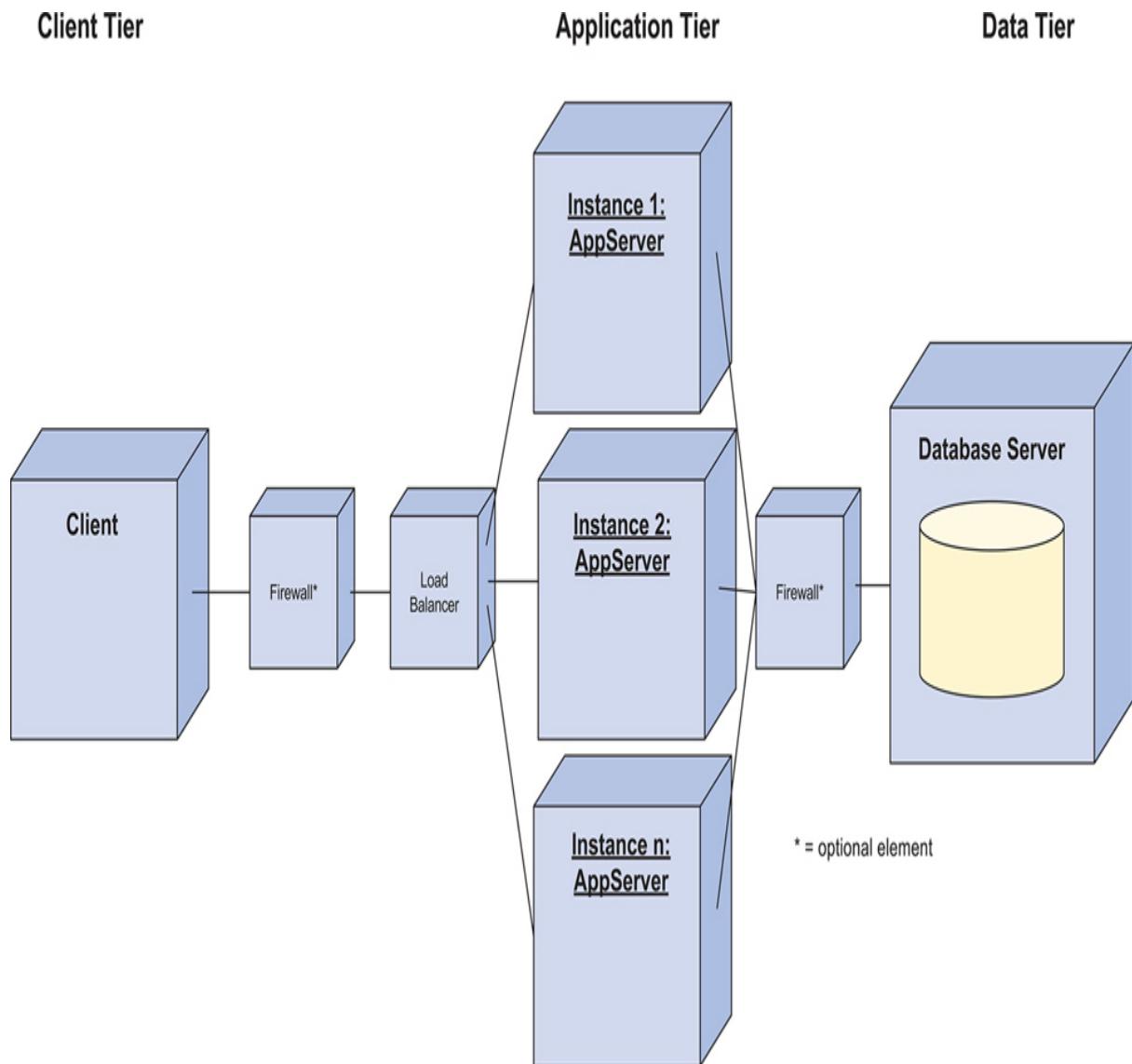


FIGURE 6.7 A Load-Balanced Cluster Deployment.

The above patterns were all ways to structure the hardware and software onto which applications were deployed. Now we turn our attention to a different class of patterns, ones that allow for different modes of deployment and rolling back deployments. Two patterns for complete replacement are well-known, both of which are realizations of the scale rollouts tactic: blue/green and rolling upgrade.

6.2.2.5 Blue/Green Pattern

In a blue/green deployment, N new instances of a Service A are created; these are termed the “green” instances. After the N instances of new Service A are deployed, the DNS server, load balancer or discovery service is changed to point to the new version. Once it is determined that the new instances are working satisfactorily, then—and only then—are the N instances of original Service A removed.

Before this cutoff point if a problem is found in the new version, perhaps via monitoring, it is a simple matter of switching back to the original (the blue services) with little or no interruption to the service clients.

6.2.2.6 Rolling Upgrade

A rolling upgrade replaces instances of Service A with instances of the new version one at a time (or a small fraction at a time). The steps of the rolling upgrade are as follows:

1. Allocate resources for a new instance of Service A.
2. Deploy the new instance.
3. Begin to direct requests to the new instance.
4. Choose an instance of the old Service A, allow it to complete any active processing.

5. Destroy that instance.
6. Repeat the preceding steps until all instances of the old version have been replaced.

The benefit of these two patterns is the ability to completely replace deployed versions of services without having to take the system out of service, thus increasing the system's availability. The cost is resources and in this dimension these two patterns differ: the peak resource utilization for a blue/green approach is $2N$ instances, whereas the peak utilization for a rolling upgrade can be as small as $N + 1$ instances.

There are other issues to consider as well. If you use blue/green deployment, then at any time a client is either using the new version or the old version. If you are using rolling upgrade, both versions are simultaneously active. This introduces two potential problems:

- *Temporal inconsistency*: In a sequence of requests by Client C to Service A, some may be served by the old version of the service and some may be served by the new version. If the versions behave differently, this may cause Client C to produce erroneous, or at least inconsistent, results. (However, this can be prevented by using the manage service interactions tactic.)
- *Interface mismatch*. If the interface to the new version of Service A is different from the old version, then invocations by clients that have not been updated to reflect the new interface will produce unpredictable results. This can be prevented by extending (but not modifying) the existing interface, or using an adapter pattern to translate the old interface to the new one.

What if you do not wish to completely replace your services? Sometimes changing all instances of a service is

undesirable. Partial-deployment patterns aim at providing multiple versions of a service simultaneously for different user groups; they are used for purposes such as quality control (canary testing) and marketing tests (A/B testing).

6.2.2.7 Canary Testing

Before rolling out a new release, it is prudent to test it in the production environment, but with a limited set of users. Canary testing is the continuous deployment analog of beta testing. Canary testing is named after the 19th-century practice of bringing canaries into coal mines. Coal mining releases gasses that are explosive and poisonous. Because canaries are more sensitive to these gasses than humans, coal miners brought canaries into the mines and watched them for signs of reaction to the gasses. The canaries acted as early warning devices for the miners, indicating an unsafe environment.

Canary testing designates a small set of users who will test the new release. Sometimes, these testers are so-called power users or preview-stream users from outside your organization who are more likely to exercise code paths and edge cases than typical users who may use the system less intensively. Users may or may not know that they are being used as guinea pigs—er, that is, canaries. Another approach is to use testers from within the organization that is developing the software. For example, Google employees almost never use the release that external users would be using, but instead act as testers for upcoming releases.

In both cases, the users are designated as canaries and routed to the appropriate version of a service through DNS or load-balancer settings or through discovery-service configuration. After testing is complete, users are all directed to either the new version or the old version, and instances of the deprecated version may be destroyed.

Rolling upgrade or blue/green deployment could be used to deploy the new version. If the canary versions are implemented using feature toggles then the rollout (or rollback) is even simpler--simply toggling the feature in the desired direction.

6.2.2.8 A/B Testing

A/B testing is used by marketers to perform an experiment with real users to determine which of several alternatives yields the best business results. A small but meaningful number of users receive a different treatment from the remainder of the users. The difference can be minor, such as a change to the font size or form layout, or it can be more significant. The “winner” would be kept, the “loser” discarded, and another contender designed and deployed. An example is a bank offering different promotions to open new accounts. An oft-repeated story is that Google tested 41 different shades of blue to decide which shade to use to report search results.

Similar to canary testing, load balancers, DNS servers and discovery-service configurations can be set to send client requests to different versions. In A/B testing, the different versions are monitored to see which one provides the best response from a business perspective.

As with the patterns for complete replacement of services, patterns for partial replacement are helped when the services being replaced are built with the tactics of encapsulation and externalization of state in mind.

6.3 Deployability and ADD

Now that we have a solid understanding of the architectural impacts of deployability, and the design options at an

architect's disposal, we turn our attention to the process of design. Specifically, we look at how ADD can be used to support design for deployability.

In what follows we assume that deployability has already been determined to be an architectural driver. This discussion of ADD focuses on the specific concerns of deployment, as they relate to the steps of the method.

Step 1. Review inputs

In this initial step one or more deployability scenarios should have been collected as part of the architectural drivers. The general scenario for deployability, as shown in [Table 6.1](#) in [section 6.1.1](#), can help here in reflection or in eliciting such scenarios from stakeholders.

Step 2. Establish iteration goal by selecting drivers

At this point we have established that deployability is important to the success of the system and have elicited one or more deployability scenarios. For this iteration, one of these scenarios will be the primary driver. Of course, other iterations could consider other deployability scenarios.

Step 3. Choose one or more elements of the system to refine

In [section 6.1.3](#) we stated that, to design for deployability, the architect needs to think about: the architecture of the application being deployed, and the architecture of the deployment infrastructure.

With respect to the architecture of the application being deployed, the granularity of deployment is one of the most important decisions that you can make and influences the overall system structure

significantly. In choosing one or more elements of the system to refine, you are choosing a granularity. Do you want to deploy big-bang style, where the entire system is a monolith that is redeployed each time? Do you want to make each function its own microservice? Or do you want something in between? This decision can be tricky as it may be affected by multiple types of requirements and concerns. If you are in a highly regulated domain, perhaps deploying safety-critical software, then you will likely opt for few releases per year and these releases will have a low granularity, perhaps releasing a single monolith each time. If you are in a highly competitive, fast-moving, minimally regulated business area (say, e-commerce or entertainment) then you might choose a much finer granularity so that individual features can be deployed or redeployed with little impact on the rest of the system. As you increase the granularity the complexity of your deployment infrastructure will necessarily increase, and you will typically want most or all of the deployment process to be automated.

Step 4. Choose one or more design concepts that satisfy the drivers

Having chosen one or more drivers and having chosen a granularity, you now need to think about what design concepts will satisfy those requirements. At this point you will be selecting and tailoring tactics for deployability as well as patterns for deploying services, patterns for partial replacement of services, or patterns for complete replacement of services.

Step 5.Instantiate architectural elements, allocate responsibilities and define interfaces

As with any design choice, the design concepts chosen in step 4 now need to be instantiated. The instantiation decisions will be of several forms. If you have, for example, chosen microservices, you need to decide how many microservices, how they are packaged, and to where they are deployed (embedded device, cloud infrastructure, hosted servers, etc.). And if you wish to automate your deployments then you will need an automation tool, such as Jenkins or GitLab.

Step 6.Sketch views and record design decisions

Deployment decisions have enormous impacts on maintainability, testability, availability, security and many other key aspects of system success. So, these decisions and the rationale for them, should be documented. All high maturity DevOps projects include a way to capture how their systems are deployed and to keep track of the state of the deployment, possibly in a Configuration Management DataBase (CMDB). Deployment diagrams could be captured using UML but they are more commonly represented using informal notations such as the ones that employ cloud provider icons (see [figure 7.2](#)). In addition, your deployment scripts are software too. These also deserve to be carefully designed, as they may outlive individuals and may become quite complex over time. Each component or service should have a deployment process. This process must define at least the following aspects:

- What artifacts are being deployed

- What action is being done to initiate the deployment
- What metrics should be monitored during the deployment (to spot problems and to potentially stop the deployment)
- How to stop the deployment
- How to roll it back

Step 7. Perform analysis of current design and review iteration goal and achievement of design purpose

Finally, your deployment decisions, because they have profound implications for many important system qualities, should be scrutinized. This might take the form of design reviews or code reviews (of the deployment scripts). Any risks or issues discovered as a result of these reviews should be added back into the backlog.

6.4 Summary

In this chapter we have described the quality attribute of deployability and shown how you can design for deployability in a disciplined, repeatable way. This involves two major sets of decisions: a set of decisions about how to structure your system (monolith, microservices, etc.) and a set of decisions about how to structure your deployment infrastructure, primarily achieved through the design of a deployment pipeline and patterns for all-or-nothing deployment or partial deployment.

Deployability, along with API centric design, form the foundations of business agility today.

6.5 Further Reading

This technical report--R. Kazman, S. Echeverria, J. Ivers, “Extensibility”, CMU/SEI-2022-TR-002, 2022--contains many of the ideas about patterns found in this chapter.

The book *Software Architecture in Practice* contains a chapter devoted to the quality attribute of Deployability: L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, 4th ed., Addison-Wesley, 2021.

A nice discussion of the principles behind microservice design, and many other related topics including an extensive catalog of design patterns, can be found at: <https://microservices.io/>

The drivers for granularity disintegration and integration come from N. Ford, M. Richards, P. Sadalage, Z. Dehghani, *Software Architecture: The Hard Parts*, O'Reilly, 2021.

For a slightly different take on deployability, this book—N. Ernst, J. Delange, R. Kazman, *Technical Debt in Practice—How to Find It and Fix It*, MIT Press, 2021—contains a chapter devoted to the problems of deployment debt, focusing on how to identify it, how to manage it, and how to avoid it.

The book *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* by Jez Humble and David Farley, Addison-Wesley, 2010 was an early treatment of the discipline of continuous engineering as that field was emerging and becoming mainstream.

Finally, a discussion of the architectural implications of DevOps can be found in: L. Bass, I. Weber, L. Zhu, *DevOps: A Software Architect's Perspective*, Addison-Wesley, 2015.

6.6 Discussion Questions

1. In [section 6.1](#) we mentioned that, to design for deployability, decisions need to be made about: 1) the architecture of the system being deployed, and 2) the architecture of the deployment infrastructure. Can ADD be used to design the deployment infrastructure? If so, how would this differ from using ADD to design the system itself?
2. Name three kinds of systems for which continuous delivery is a bad idea. Which of the tactics and patterns would you not use for such systems and why?
3. In [Section 6.1.3](#) we claimed that an architect worrying about deployability needs to be concerned with the degree to which an architecture is granular, controllable, and efficient. How would you go about determining the “right” granularity for the pieces of an architecture that you wished to deploy?
4. What business drivers would lead you to choose a blue/green deployment pattern versus a rolling upgrade deployment pattern? What are the pros and cons of each?
5. What business drivers would lead you to choose canary testing versus A/B testing? What are the pros and cons of each?
6. What are the challenges in replicating a microservices architecture across pre-production environments? Consider that there may be many dependencies, to third party applications or infrastructure resources.

7. Designing Cloud-Based Solutions

In this book we talk about software architecture and how it allows drivers and particularly quality attributes to be satisfied. For many software-intensive systems, certain quality attributes cannot be satisfied purely through software design decisions. They can only be satisfied through combinations of software and infrastructure design decisions. In this chapter we discuss how designing the infrastructure and, more specifically a cloud infrastructure, is also an essential part of the design of the architecture.

Why read this chapter?

Having read this chapter you should not expect to become an expert on cloud-based architectural solutions, as this is an extensive set of technologies, and its precise details are covered elsewhere. Rather we hope you will understand the types of drivers and the spectrum of design concepts associated with cloud development. Here we focus on how these design concepts can be employed in the context of ADD. As our case studies both employ cloud-based infrastructures, a knowledge of these design concepts will be key to understanding them.

7.1 Introduction to the cloud

The term Cloud Computing was defined in 1997 but it was not until 2006 that a cloud provider started offering resources to the public and, since then, cloud adoption has become widespread.

7.1.1 What is cloud computing

Cloud computing allows computer system resources to be used without the need to manage hardware. Prior to the emergence of the cloud, setting up an infrastructure typically meant interacting with physical hardware, possibly with considerable effort. Today, setting up a cloud infrastructure means selecting and configuring virtual hardware resources without having to worry about physical hardware.

There are different deployment models for cloud computing: private, public and hybrid. A private cloud consists of an infrastructure that is owned and managed by an organization for internal usage. Private clouds are typically used to comply with security and privacy regulations or to leverage existing infrastructure. A public cloud consists of an infrastructure where resources can be consumed by any customer and hybrid clouds combine private and one or more public cloud resources. In this chapter we focus primarily on resources offered by public cloud providers such as Amazon (AWS), Google (GCP), or Microsoft (Azure).

The cloud has been adopted widely because there are many benefits to consuming virtual resources rather than actually buying and installing physical hardware. Some of the benefits are the following:

- Having your own dedicated spaces to host hardware infrastructure is costly and not necessary anymore.
- Resources can be obtained and paid for only when they are needed (pay-per-use) and they can be scaled rapidly in a manual or automatic way (elastic scaling).
- Replicating the infrastructure to support the different deployment environments (integration, staging, production, see [6.1.1](#)) can be achieved in a simpler way.
- Reconfiguring the infrastructure becomes simple, as compared with a physical reconfiguration.
- Cloud providers employ experts in areas such as security and physical infrastructure that small companies could not afford if they were to have their systems hosted on premise.
- When using resources managed by the cloud provider, as discussed in the next section, it is not necessary to worry about updating the hardware or the operating system.
- Outsourcing infrastructure needs enables a business to focus resources and investments on key differentiators.

7.1.2 Service models

Cloud providers offer different models of resource provision, which are commonly called service models. Each one is characterized by the amount and type of resources that are managed by the customer. And each of these models is built upon the previous one:

- *IaaS (Infrastructure as a service)*: In this model, the cloud provider manages aspects such as virtualization, physical servers, storage and networking and the customer obtains virtual hardware and is responsible for

managing many aspects such as the operating system and everything that is executed on top of it, including runtime environments, middleware, data and applications.

- *PaaS (Platform as a service)*: In this model, the cloud provider manages the infrastructure, but also additional aspects including the operating system, runtime environments and middleware. The customer obtains platforms, but still manages their data and applications. These environments can support concerns such as elastic scaling, fault tolerance and health monitoring.
- *FaaS (Function as a service)*: In this model, the cloud provides manages most aspects and provides execution environments where functions (and not whole applications as in PaaS), developed by the customer, are executed. This is considered one variant of serverless computing as billing occurs only during the time that the functions are executed. As in PaaS, the execution environment supports aspects such as elastic scaling and fault tolerance.
- *SaaS (Software as a service)*: In this model, the cloud provider manages all aspects, including applications, and the customer pays to use the application. An application can be accessed from its user interface or can be integrated and used by calling its APIs. Third parties may offer applications as SaaS offerings; the application is managed by the third party but is deployed on a cloud provider's infrastructure.

Each of these models of resource provision involves different levels of costs, the effort that the resource consumer must devote to managing the resources and also the level of dependence on the cloud provider (vendor lock-in). It should be noted that a particular system may combine approaches, for instance, some parts of the system can be

implemented using FaaS capabilities, and other parts can use SaaS and PaaS capabilities.

7.1.3 Managed resources

Resources that are managed by the cloud provider (we will refer to them as *managed resources*) do not require the consumer to launch a virtual machine and to install software associated with the desired resource such as a database engine or a message queue. Managed resources simply need to be selected and instantiated in the cloud environment. Some of these capabilities exist across cloud providers, such as a MySQL database engine, while others are specific to the provider, for example a user management capability.

Managed resources provide different benefits including the fact that they can be launched simply and do not need to be updated manually. Also, they are well integrated with other tools from the cloud provider that support aspects such as monitoring, logging, security or backups. Furthermore, every cloud provider includes mechanisms that allow these resources to be scaled elastically to support performance needs, and replicated to support availability. It is important to note that the software that runs on elastic managed resources must be designed to support this capability (for example, by considering aspects such as statelessness).

There are, however, some downsides to using managed resources, the most important being that the application is tied to the particular cloud provider, which limits moving it to a different provider (vendor lock-in), and that these managed resources are typically costlier, in terms of billing, than using open source solutions directly on virtual machines. Furthermore, managed resources may be limited to certain options only, for example, a relational database

resource may only be instantiated from a limited selection of database engine types and versions, although these tend to be the most recent and popular.

Cloud providers expose APIs that allow resources to be managed and provisioned by interacting with the API. This approach allows infrastructure to be treated as code, as resources can be managed and provisioned through definition files that are processed by a computer, instead of having to configure them manually. This approach, referred to as infrastructure as code, facilitates for example the replication of the infrastructure across the staging and production environments.

7.2 Drivers and the Cloud

In this section we describe the architectural drivers, more specifically the quality attributes and constraints, which have a direct relationship with the use of cloud infrastructures.

7.2.1 Quality attributes

Certain quality attributes cannot be satisfied purely through software decisions. Here we describe the most common quality attributes that require a combination of software and infrastructure decisions. In [section 7.3.1](#) we go into more details about the capabilities offered by cloud providers and in [section 7.3.2](#) we describe how tactics associated with these quality attributes can be realized by these capabilities.

7.2.1.1 Performance and scalability

The cloud facilitates the modification of compute resources to increase aspects such as CPU, memory or storage to

achieve vertical scaling as well as the replication of resources to achieve horizontal scaling. Scaling can be performed manually or automatically (which is called elastic scaling), depending on the model of resource provision. Furthermore, resources can be placed in close geographic proximity or even in the same data center to reduce latency in communications (see [section 7.3.1.1](#)).

7.2.1.2 Availability

One of the key decisions to support availability is the replication of resources. Cloud infrastructure facilitates the replication of resources and also their instantiation in different geographic locations, to avoid failures due to natural disasters. Cloud providers also supply tools that facilitate the monitoring of resources so that actions can be taken in case of failures. It should be noted that there is a tradeoff between distributing resources across geographical locations to increase availability and performance.

7.2.1.3 Security

Security is a quality attribute that requires many measures to be taken at the infrastructure level, and often these measures can be quite complex to create and maintain. Cloud providers typically offer mechanisms to create virtual private networks and also offer capabilities such as firewalls, authorization and authentication servers, DDoS protection services, gateways and many others to address this quality attribute.

7.2.1.4 Deployability

Deployability, as we saw in [chapter 6](#), is the capability of being able to continuously, predictably, and with minimal effort and human intervention, deliver new versions of a system or system component. Delivery involves deploying

the code across the different pre-production environments until it is deployed into the final production environment. Cloud infrastructures provide capabilities to set up and execute deployment pipelines along with capabilities to replace services partially or completely and to rollback deployments in case of issues.

7.2.1.5 Operability

Operability is the ability to keep a system in a safe and reliable functioning condition, according to pre-defined operational requirements. An important part of operability is the capacity of the system to support observability and monitoring to detect and react when problems occur. Cloud providers offer capabilities to monitor resources so that they can operate adequately.

7.2.2 Constraints

In the following sections we discuss important constraints associated with the use of cloud infrastructures.

7.2.2.1 Cost optimization

Using cloud resources has an associated cost. Each additional capability that is used contributes to the cost of the infrastructure. Furthermore, the particular characteristics of the capabilities also contribute to the cost (e.g. CPU, memory and storage sizes, managed or not, and the pay per use model).

Cost is always a constraint in development, so it is important to be able to estimate how much a particular solution will cost and to evaluate the cost of alternative solutions. Cloud providers offer calculators (see [figure 7.1](#)) which allow costs to be estimated for a given infrastructure.

These tools can be extremely useful during the design process to help make decisions that optimize costs.

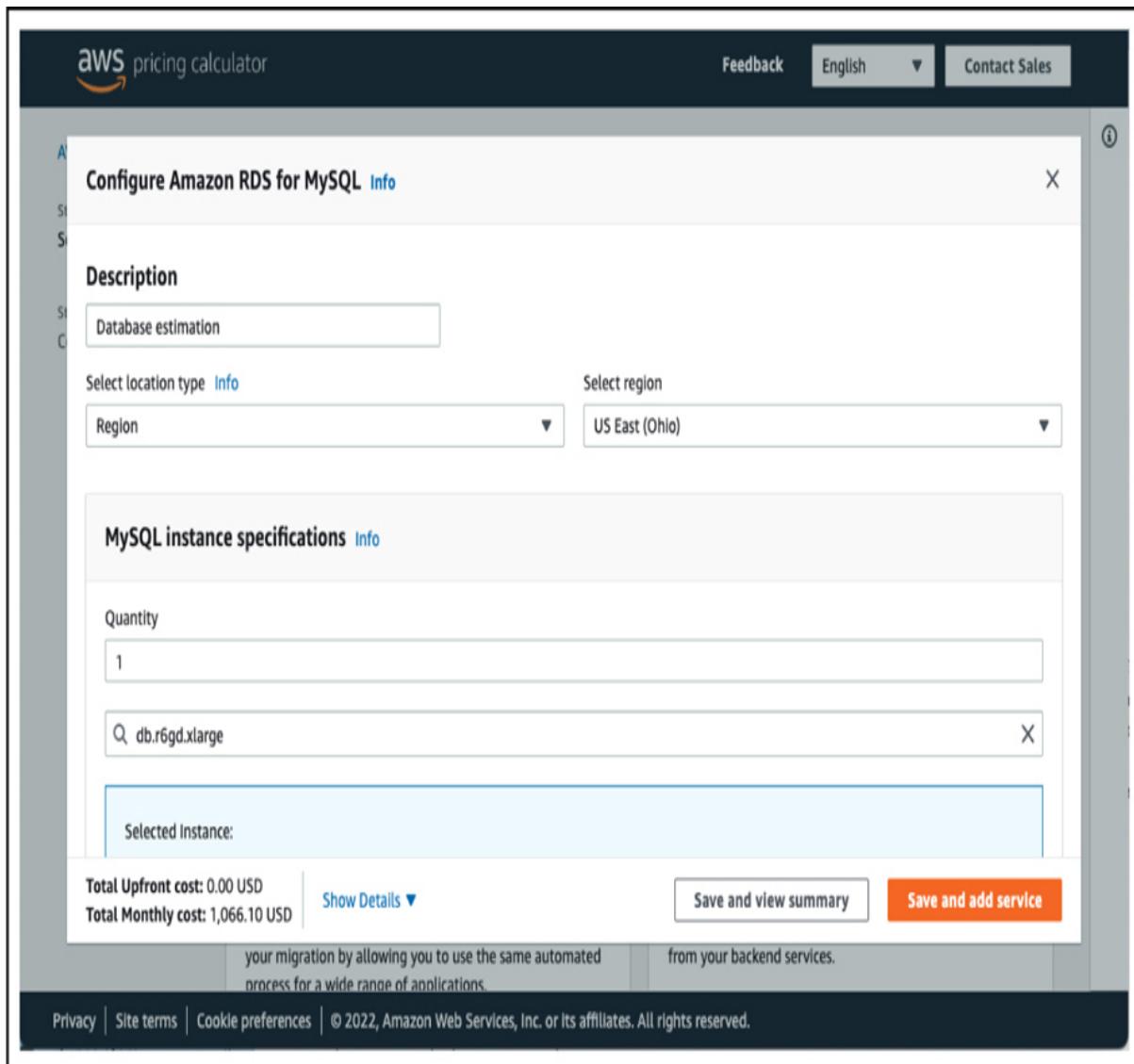


FIGURE 7.1 Cloud provider cost estimation calculator

Once deployed, the actual costs of operation need to be monitored and optimized if necessary. Some measures can be taken to reduce costs such as shutting down integration environments during the weekends or using less expensive resources than the ones used in the staging or production environments.

7.2.2.2 Cloud native and vendor lock-in

In the last decade, many companies have started to migrate their enterprise IT infrastructure to the cloud. This can be a major effort as many of the applications that are used by the company may have been designed in an era where the cloud did not exist or was not so prevalent. One relatively simple approach of moving to the cloud is the so-called “lift and shift” approach. This involves moving an application which runs on-premise to the cloud, without making significant changes to the application. This approach is usually associated with the IaaS model of resource provision as it typically involves obtaining cloud infrastructure resources similar to the ones used on-premise. This approach, while simple, does not make as effective use of the cloud resources as possible, but it can be a first step in moving to the cloud.

A different approach involves re-writing parts or the whole application to make effective use of cloud resources. This results in an application that is referred to as “cloud native”. When building cloud native applications, they are built using a cloud-specific approach, that is they use managed resources or a platform specific to a cloud provider. Choosing a cloud native strategy can be a project constraint that simplifies the choices of resources, as only the resources offered by the cloud provider need to be considered.

An opposite constraint may be to avoid vendor lock-in by following a cloud-agnostic approach. The customer may require that the system is not tied to a particular cloud provider and that it can easily be moved to a different provider if necessary. This constraint limits the kinds of resources that can be consumed and the models of resource provision that can be employed. When this constraint exists, the solution typically is to avoid vendor-specific managed

resources, which may be replaced by open source products deployed on the cloud, or to use IaaS as the model of resource provision.

7.2.2.3 Legal considerations

The data centers where the cloud resources reside are scattered across the globe. However, there may not be a data center for a given cloud provider in every country. This means that a given application and its data cannot always reside in a particular country. This may violate legal requirements about sensitive information from one country being hosted in a different one or being hosted by a different company.

Furthermore, there may be some mistrust in the handling of information and security measures from cloud providers which further limit the desire to host an application in a public cloud.

7.3 Cloud-Based Design Concepts

The design concepts used in the construction of cloud-based solutions include a number of capabilities along with tactics and patterns realized by these capabilities. In this section we present these design concepts.

7.3.1 Externally developed components: cloud capabilities

While there is a variety of cloud providers, most of them offer a similar set of fundamental cloud capabilities. Cloud capabilities can be considered as one type of externally developed components. The most common capabilities are presented in the following subsections.

7.3.1.1 Regional capabilities

Cloud resources are hosted in data centers that are scattered around the globe. Cloud providers typically organize resources hierarchically as follows:

- *Regions*: These are separate geographic areas, typically grouping one or more data centers. An example of a region could be the Western United States.
- *Availability zones*: These typically correspond to specific data centers within a region. Applications can be partitioned across availability zones so that a failure in a particular data center does not result in a failure of the application.
- *Localized capabilities*: These correspond to resources that are hosted in geographical locations that are close to the users of those resources, for example on the edge of telecommunication carrier's networks or even to on-premises locations.

7.3.1.2 Computing capabilities

The most essential capability that is supplied by cloud providers is a set of computing resources. These include:

- *Virtual machines*: These are virtual servers in the cloud which are configured with a particular operating system, a particular hardware configuration (memory, CPUs, hard drive capacity, etc...). These are typically associated with the IaaS resource provision model.
- *Web application execution environments*: These are execution environments for applications written in different programming languages. Configuration options typically include the particular platform (for example, Java or .NET) and some specific options for the given

platform. These are typically associated with the PaaS resource provision model.

- *Serverless execution environments*: These are execution environments for functions which are executed as a service. As opposed to virtual machines, these environments do not require a particular hardware configuration to be guaranteed.
- *Container execution environments*: These are execution environments for containers (which include container orchestrators). The container image specifies the operating system and some hardware configurations can be established such as the amount of CPU and memory that are provided to the container. These environments can support concerns such as automatic provisioning, scaling and health monitoring.

Some of these execution environments, specifically those that are not IaaS, support automatic scaling and other features such as integration with security and logging capabilities.

7.3.1.3 Networking capabilities

Cloud providers also supply networking capabilities that are required to connect other capabilities such as computing and storage. Examples of these networking capabilities include::

- *Virtual Private Networks (VPNs)*: these are virtual networks where cloud capabilities used in the application reside. A VPN can be associated with a region. Subnets are ranges of IP addresses which are created inside VPNs and which can be associated with availability zones. Subnets can be public for resources that are connected to the internet or private, for resources that are not.

- *Domain Name Servers (DNS)*: resources that are created inside VPNs are associated with names that are resolved using domain name servers.
- *Gateways*: These connect VPNs to other networks, including the internet. API Gateways (see section 5.3.1.4) are a specific type of Gateway that allows APIs to be published, secured, monitored and maintained.
- *Content Delivery Networks (CDNs)*: These are servers that are distributed geographically and which help deliver data to users with reduced latency by delivering the data from a server that is close and well connected to the end user.
- *Service Meshes*: This is an infrastructure layer that manages communication between microservices. The service mesh introduces proxies called sidecars that handle aspects such as communication or monitoring.

7.3.1.4 Storage capabilities

Cloud providers supply capabilities that allow information to be stored inside file systems. Examples of these capabilities include:

- *Instance storage*: This is a storage capability that is associated with a virtual machine, similar to a hard drive for the virtual machine.
- *Network File Systems (NFS)*: These are managed file systems which can be shared across multiple instances and which are able to scale on demand.
- *Object storage*: This is a scalable mechanism that stores objects but they are not stored hierarchically, in contrast to file systems.
- *Backup services*: These are services that allow backups for different resources to be created and managed.

7.3.1.5 Database capabilities

Information storage inside databases is another important capability that is universally provided. Examples of these capabilities include:

- *Managed relational databases*: These are relational databases that are managed by the cloud provider, using different engines (such as MySQL or PostgreSQL). Hardware, availability, reliability, backups, security and monitoring characteristics can be configured.
- *Managed NoSQL databases*: These are non relational databases of different varieties: Document, Key-Value, Graph, Column, and Time series.
- *In-Memory caches and Databases*: These are in-memory data stores and cache services.

7.3.1.6 Development and DevOps capabilities

Cloud providers also supply capabilities to support software development. These range from code repositories to CI/CD pipelines that facilitate DevOps. Examples of these capabilities include:

- *Code repositories*: These are managed version control systems, most commonly Git repositories.
- *Artifact repositories*: These are managed repositories for artifacts which facilitate the storage, publication and sharing of software packages.
- *Pipelines*: These are managed continuous integration and deployment/delivery servers which automate the build, test and release process.
- *Deployment services*: These are services that manage deployments to different environments.

- *Analysis and debugging tools*: These are tools that facilitate analysis and debugging of distributed systems, for example using distributed tracing.
- *Infrastructure as code services*: These services allow physical architectures to be described using a domain-specific language. These descriptions can be used to easily set up the infrastructure across environments, without the need of instantiating and configuring all of the cloud resources manually.
- *Monitoring services*: These services collect data from different sources including logs and events. They also provide mechanisms to visualize and analyze logged data using dashboards, and to set up alarms and actions based on thresholds.

7.3.1.7 Security capabilities

Security is a fundamental quality attribute that requires design decisions to be made both at the software and at the hardware level. At the hardware level, cloud providers supply many security capabilities, such as:

- *Authentication and authorization services*: These are services that allow users to be managed and that provide access tokens to grant access.
- *Firewalls*: These are network protection mechanisms that are deployed to protect VPNs.
- *Certificate managers*: These are services that allow SSL/TLS certificates to be provisioned, managed and deployed.
- *Secret managers*: These are services that allow sensitive information (such as passwords) to be managed.

- *Threat detection services*: These are services that identify threats such as malicious usage of compute capabilities or suspicious activity in storage capabilities.
- *DDoS protection services*: These are services that are specifically designed to protect against this particular type of attack.

7.3.1.8 Application integration capabilities

Application integration is essential to support the construction of distributed systems or to allow different applications to communicate with each other. Cloud providers offer several services to support application integration such as the following.

- *Message queues*: These are services that allow message brokers to be provisioned and operated.
- *Notification services*: These are services that allow publisher/subscriber messaging mechanisms to be provisioned and operated.
- *Event streaming services*: These are services that allow event streaming platforms to be provisioned and operated.

7.3.1.9 Analytics capabilities

Operational data is the data that supports the day-to-day operation of the business and which is typically stored in databases. Analytical data, on the other hand, is used to support business intelligence to make long term strategic decisions and give direction. Some capabilities that support analytics are:

- *Data lakes*: These are services that allow centralized repositories that store raw structured and unstructured

data to be created and managed. Data is ingested and cleansed before being stored securely.

- *Data warehouses*: These are services that allow central repositories of curated structured information used for analysis to be created and managed. Data is received from databases, transactional systems and other sources and stored securely.
- *Search and query services*: These are services that facilitate performing searches and queries on large volumes of information.

7.3.1.10 Additional capabilities

Cloud providers offer additional capabilities that support common development needs. The following are some of the most popular ones.

- *Machine learning capabilities*: These are services that facilitate the construction, training and deployment of machine learning models.
- *Blockchain capabilities*: These are services that allow blockchains to be created and managed.
- *Internet of things capabilities*: These are services that support the development of IoT applications, including aspects such as deployment and execution of code on devices, connection of devices to the cloud and device management.
- *Game development*: These are services that facilitate the development of games, particularly large multi-player games.

7.3.2 Tactics

In this section we review the tactics that are associated with the quality attributes presented in [section 7.2.1](#) and show how they can be achieved using the capabilities that were discussed previously.

7.3.2.1 Performance and scalability

Some performance tactics (see [section 3.3.1](#)) that are supported by cloud capabilities are under the resource management group and they include:

- *Maintaining multiple copies of data / Maintaining multiple copies of computation*: cloud capabilities such as virtual machines or databases can easily be replicated and capabilities such as load balancers can be used to distribute requests across replicas. Managed databases typically handle replication of data automatically.
- *Increasing resources*: several cloud capabilities can be scaled both horizontally and vertically. For example, a virtual machine with a given CPU and memory configuration can easily be changed to a more powerful CPU with a larger memory allocation.

7.3.2.2 Availability

Availability tactics (see [section 3.4.1](#)) that are supported by cloud capabilities include:

- *Condition monitoring*: Certain cloud capabilities can be monitored by the cloud infrastructure, for example, a container execution environment. In case of failures, a new container instance can be launched automatically.
- *Redundant spares*: Replicating capabilities across availability zones or geographical regions support the realization of this tactic. The cloud infrastructure

provides mechanisms to ensure that replicas maintain consistency (for example in databases).

- *Software upgrades*: The cloud infrastructure allows non-service-affecting upgrades to be performed. New compute capabilities with updated versions of the software can be launched while previous versions still process requests. Once the updated versions are running, traffic can be directed to them.
- *Reintroduction*: Reintroducing faulty components is also facilitated by the cloud infrastructure. For example, launching a container after a failure is relatively easy and it can be handled by the execution environment.
- *Rollback*: Certain types of rollbacks, such as deployments, can be handled by the cloud infrastructure.

7.3.2.3 Security

Security is a quality attribute that is extensively supported by cloud capabilities. Security tactics (see [section 3.6.1](#)) that can be partially or fully implemented using cloud capabilities include the following:

- *Detection of intrusions*: Cloud providers offer services to detect intrusions or malicious usage of resources.
- *Detection of denial of service*: DDoS protection services are specifically designed to guard against this type of attack.
- *Authentication of actors*: Authentication and authorization services support the implementation of this particular tactic.
- *Limiting access*: Limiting access to resources can be achieved through different mechanisms such as the use of VPCs or API Gateways.

- *Data encryption*: Capabilities such as databases offer the possibility of encrypting data at rest, while data that is transferred can also be encrypted by using protocols such as TLS.
- *Informing actors*: Notification mechanisms support informing actors when malicious activity is detected by capabilities such as threat detection services.
- *Auditing*: Monitoring and storage capabilities support the collection, storage and further analysis of information including audit logs.

7.3.2.4 Deployability

Deployability tactics (see [section 6.2.1](#)) include tactics related to managing a deployment pipeline and managing the deployed system. Cloud providers offer a number of capabilities that allow the deployment pipeline to be managed.

- *Scaling rollouts*: Cloud providers offer services that support the capability of scaling rollouts.
- *Scripting deployment commands*: Infrastructure as code capabilities are offered by cloud providers, either through proprietary mechanisms or by supporting popular open source solutions.
- *Rollbacks*: As with scaling rollouts, services that support rollbacks of deployments are also provided.

7.3.2.5 Operability

While there is not a catalog of tactics to support operability, aspects such as monitoring and observability are supported by capabilities offered by cloud providers. Monitoring services, analysis and debugging tools and dashboards allow the collection of operational information, including

logs, and its visualization to support health analysis. Alarms and notifications can be configured to identify problematic situations. The information that is collected is also essential for optimizing costs by identifying, for example, unused or underused resources. Supporting observability also requires certain design decisions to be made in the software architecture, for example the integration of libraries that allow metrics to be collected.

7.3.3 Patterns

There are numerous resources that provide information on patterns associated with cloud-based development.

7.3.3.1 Reference architectures

Reference architectures provide blueprints for developing specific types of applications. There are reference architectures that specifically employ cloud capabilities to solve development and operation problems. Reference architectures for cloud-based applications are most commonly associated with a specific cloud provider's capabilities, but they can easily be translated to a different cloud provider with equivalent capabilities. [Figure 7.2](#) illustrates an example of a typical microservices application on AWS.

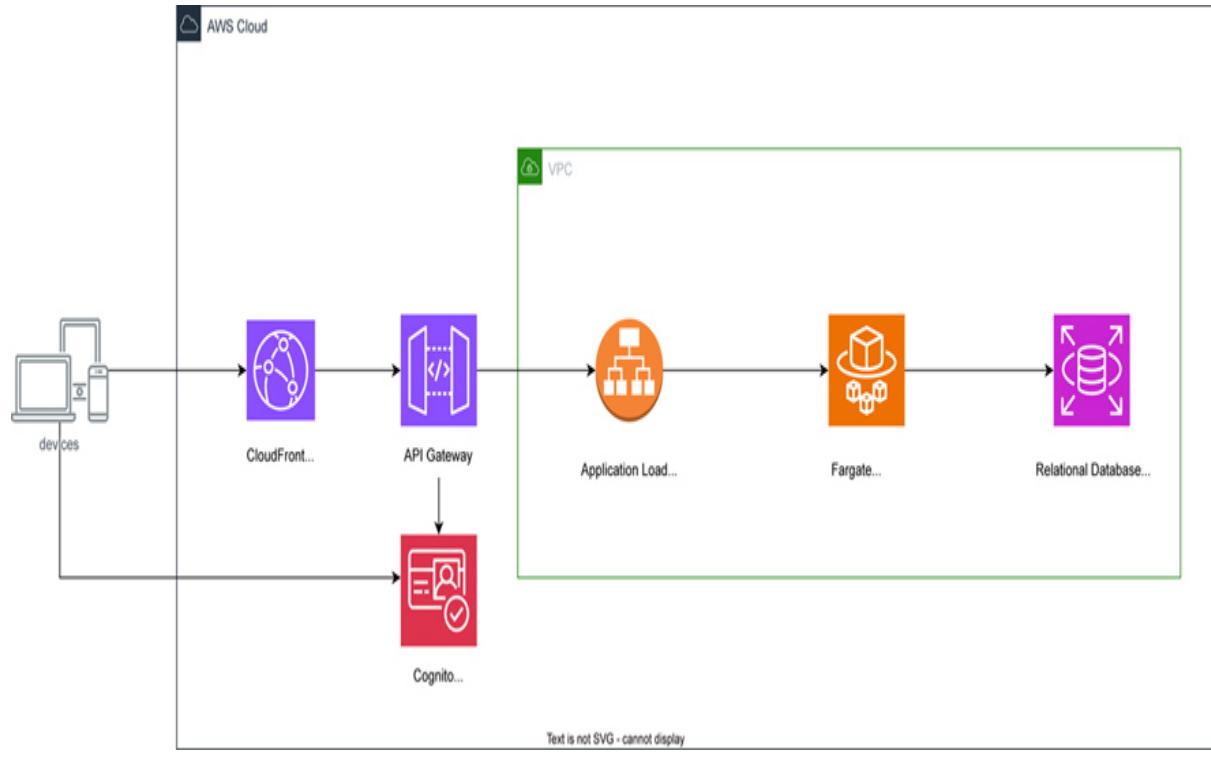


FIGURE 7.2 A simple microservice application on AWS

7.3.3.2 Design patterns

There are many patterns that help achieve the quality attributes that are supported by the use of the cloud: performance, scalability, availability and security, among others.

An example of these patterns is CQRS, which means Command and Query Responsibility Segregation. In this pattern, the update (command) and read (query) operations are separated and handled by different components. Changes that result from updates are propagated from the command side to the query side using mechanisms such as events. The command and query components can use different data stores (for example, a relational database on the command side and a document database on the query side). Furthermore, the command and query sides can be scaled independently. This pattern promotes availability,

performance and security. Availability is promoted since a failure on the command side does not affect the query side, and vice versa. Performance is promoted as the query side can receive data from multiple sources, allowing data from different system or domain boundaries to be queried efficiently. Also, the query side can be scaled independently from the command side and querying the data does not affect the performance of the command side. Finally, security is promoted since access to the command side can be restricted and clients that have access to the query side cannot perform updates. The disadvantage of this pattern is that there is some complexity associated with its implementation and also, the separation of the database brings up eventual consistency concerns.

While the CQRS pattern is cloud provider agnostic, it can be implemented using capabilities from a specific cloud provider, as illustrated in [figure 7.3](#).

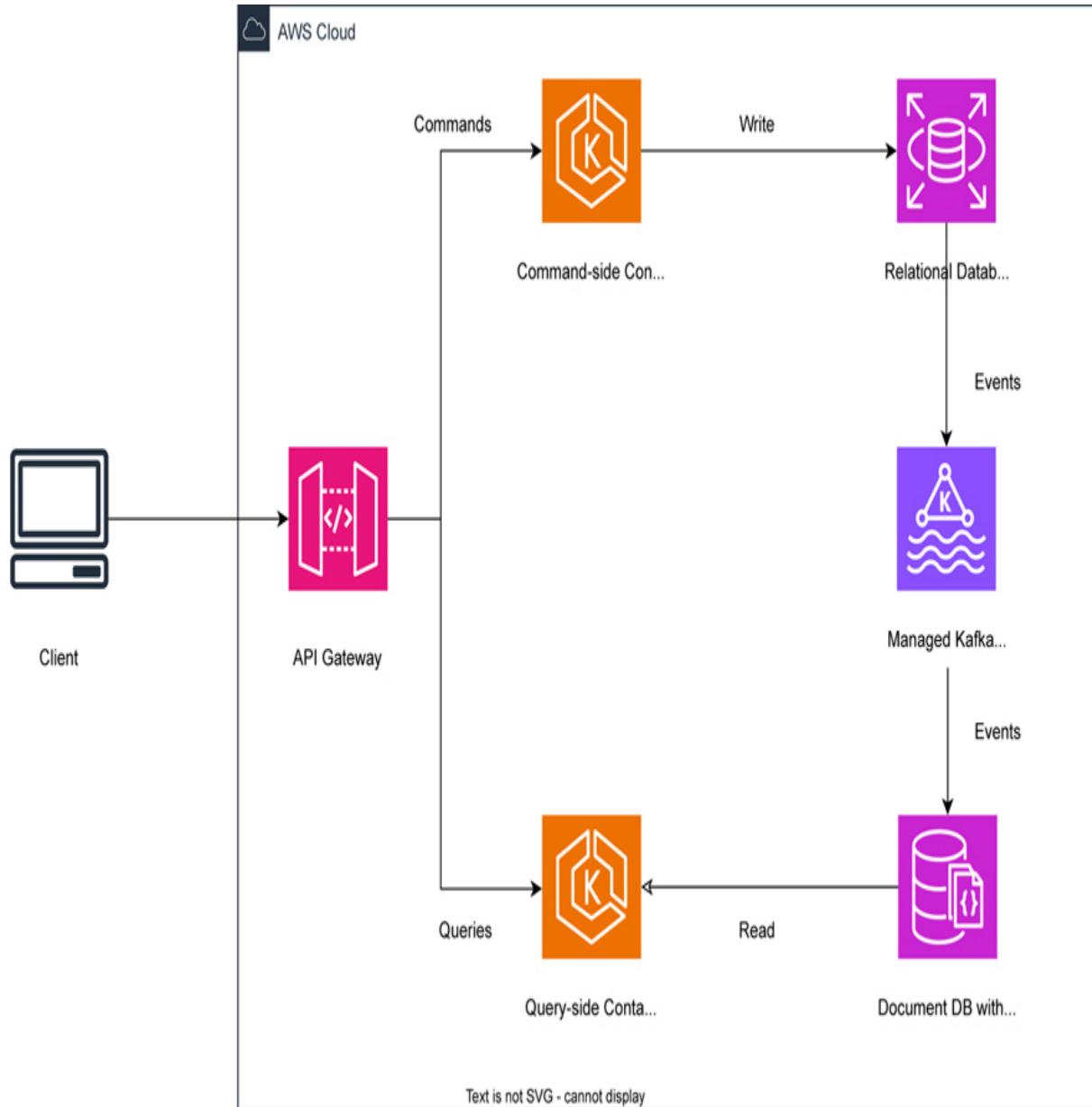


FIGURE 7.3 Implementation of CQRS using AWS capabilities.

There are other resources, including books on design patterns for cloud computing. Please see the further reading section of this chapter for additional references.

7.4 ADD in the design of cloud-based solutions

Here we discuss considerations that can be made in each of the steps of ADD that are related to the use of a cloud infrastructure.

Step 1. Review inputs

In this step, it is essential to pay attention to the constraints associated with the use of a cloud infrastructure. These constraints, which may be defined by an organization's enterprise architecture team, may mandate the use of resources from a particular provider, or the need to avoid vendor lock-in, or legal restrictions which limit the geographic region where the application and its data can reside. Also, clarity with respect to budget constraints for implementation and operation is essential as this may guide design decisions along the way.

Step 2. Establish iteration goal by selecting drivers

One common early iteration goal is to design the infrastructure of the application; in this case the drivers that are selected should be quality attributes that must be satisfied through combinations of software and infrastructure decisions as well as previously identified constraints.

In addition, as products are continuously evolved, changes in the existing infrastructure may be required. In this case, the iteration goal may be to refine a previously defined infrastructure.

Step 3. Choose one or more elements of the system to refine

The elements to refine are typically software elements and the refinement involves the identification of infrastructure elements where these software elements will reside. For example, a

previous iteration may have produced a structure that involves microservices. In this iteration these microservices are identified and the refinement will involve the selection of capabilities where these microservices will be executed and where data will be stored.

If the iteration goal is to refine a previously defined infrastructure, the elements that are selected can be infrastructure components that were identified in this previous iteration.

Step 4. Choose one or more design concepts that satisfy the drivers

During initial iterations, design decisions may involve choosing the type of cloud (public, private, hybrid), selecting the resource provision model (or models) that can be used (IaaS, PaaS, etc.) and selecting a reference architecture. These decisions can be guided by the approach used to structure services (e.g. monolith, microservices), the workload that is expected in the application, the type of data that needs to be managed, and how these are expected to scale in the future. Subsequent iterations will involve the selection of capabilities that are aligned with the decisions made previously.

Design concepts selected at the infrastructure level must be aligned with the decisions made at the software level. For example, selecting a managed elastic compute capability requires that the software that runs on these compute capabilities can be replicated (by decoupling the application state from the life cycle of the compute resources, for example).

One benefit of using the cloud is that it is easy to instantiate capabilities to perform experiments and prototypes. These activities can be useful in the selection process. This process will usually be constrained by cost limitations; in this case it can be helpful to make use of cost calculators to guide decision-making.

Step 5. Instantiate architectural elements, allocate responsibilities and define interfaces

The instantiation of cloud capabilities involves deciding on aspects that are specific to a provider's configurations. For example, for a database capability, instantiation may involve the selection of a particular database engine (e.g. MySQL or Postgres) and the selection of values for its configuration, for example the size of the database (e.g. Small, Medium or Large instance) and configurations to support quality attributes such as security or availability.

Step 6. Sketch views and record design decisions

Cloud infrastructure diagrams are usually represented using informal notations that employ cloud provider-specific icons, as exemplified in [Figure 7.2](#). Design decisions may be recorded separately using a template for architectural decisions.

Step 7. Perform analysis of current design and review iteration goal and achievement of design purpose

As is usual in ADD, the last step of the process involves analyzing whether the design decisions that have been made up to this point are sufficient. This analysis may lead to further iterations to refine the design.

7.5 Summary

In this chapter we have discussed the design of solutions that involve an infrastructure hosted in a public cloud. Today, cloud providers offer a wide assortment of capabilities that can be extremely useful in accelerating development and satisfying quality attributes that involve software and infrastructure decisions.

There are many benefits of using the cloud and its widespread adoption is evidence of this. However, design decisions when using a cloud infrastructure must be carefully made because of their associated costs and because the use of the cloud may not be acceptable in all contexts.

7.6 Further Reading

Cloud provider websites provide a wealth of information regarding the capabilities they offer. Major cloud provider sites include:

- Amazon Web Services: <https://aws.amazon.com/>
- Google Cloud: <https://cloud.google.com/>
- Microsoft Azure: <https://azure.microsoft.com/en-us/>

And an online catalog of cloud design patterns is available at <https://learn.microsoft.com/en-us/azure/architecture/patterns/>. This catalog is hosted by Microsoft, but the patterns are independent from any specific provider.

The following book describes many important concepts associated with the development of applications that are designed as collections of microservices hosted in the cloud: Indrasiri K, Suhothayan S., “*Design Patterns for Cloud Native Applications*”, O'Reilly 2021

This book discusses many concerns that must be addressed in the construction of distributed architectures which are typically deployed in the cloud and provides guidance on the design decisions used to address them: Ford, N., Richards, M., Sadalage, P., Dehghani, Z., “*Software Architecture: The Hard Parts: Modern Trade-Off Analyses for Distributed Architectures*”, O'Reilly, 2021

7.7 Discussion Questions

1. Using the IaaS resource provision model may involve lower billing costs from the cloud provider. What other costs, not billed by the cloud provider, can be incurred by the resource customers when using this model?
2. Using infrastructure as code, the infrastructure can easily be replicated in the integration, test, and production environments. Should these three infrastructures be exactly the same?
3. Can you think of, and provide examples of, measures that can be taken to reduce costs when establishing different execution and test environments?
4. Can you think of a situation that involves (or requires) mixing resources from two or more cloud providers?
5. What important architectural decisions are orthogonal to (that is, unaffected by) the choice of cloud provider?
6. What other quality attributes beside the ones discussed in this chapter can be addressed as combinations of software and infrastructure design decisions?
7. A company wants to move a monolithic web-based application to the cloud. What are some of the challenges they may face to make optimal use of the cloud infrastructure?

8. Case Study - Hotel Pricing System

In this chapter, we present a case study of using ADD for a greenfield system in a mature domain. This case study presents the early design rounds composed of 4 iterations and is based on a real-world example. We first present the business context for the system and then we summarize its requirements. This is followed by a step-by-step summary of the activities that are performed during the ADD iterations.

Why read this chapter?

People learn best from examples. This chapter will help you to contextualize and visualize how the many concepts that have been previously introduced can be put together. We do this via a detailed example of addressing a real-world problem--a hotel pricing system--with an architectural solution.

8.1 Business Case

AD&D Hotels is a mid-sized business hotel chain (currently around 300 hotels) which has been experiencing robust growth in recent years. The IT infrastructure of the company

is composed of many different applications such as a Property Management System, a Commercial Analysis System, an Enterprise Reservation System, a Channel Management System and, at the center of this system of systems, is the Hotel Pricing System, as seen in the context diagram shown [Figure 8.1](#). While part of this system was already deployed in the cloud, the approach followed was lift and shift and thus these systems were not making full use of cloud resources.

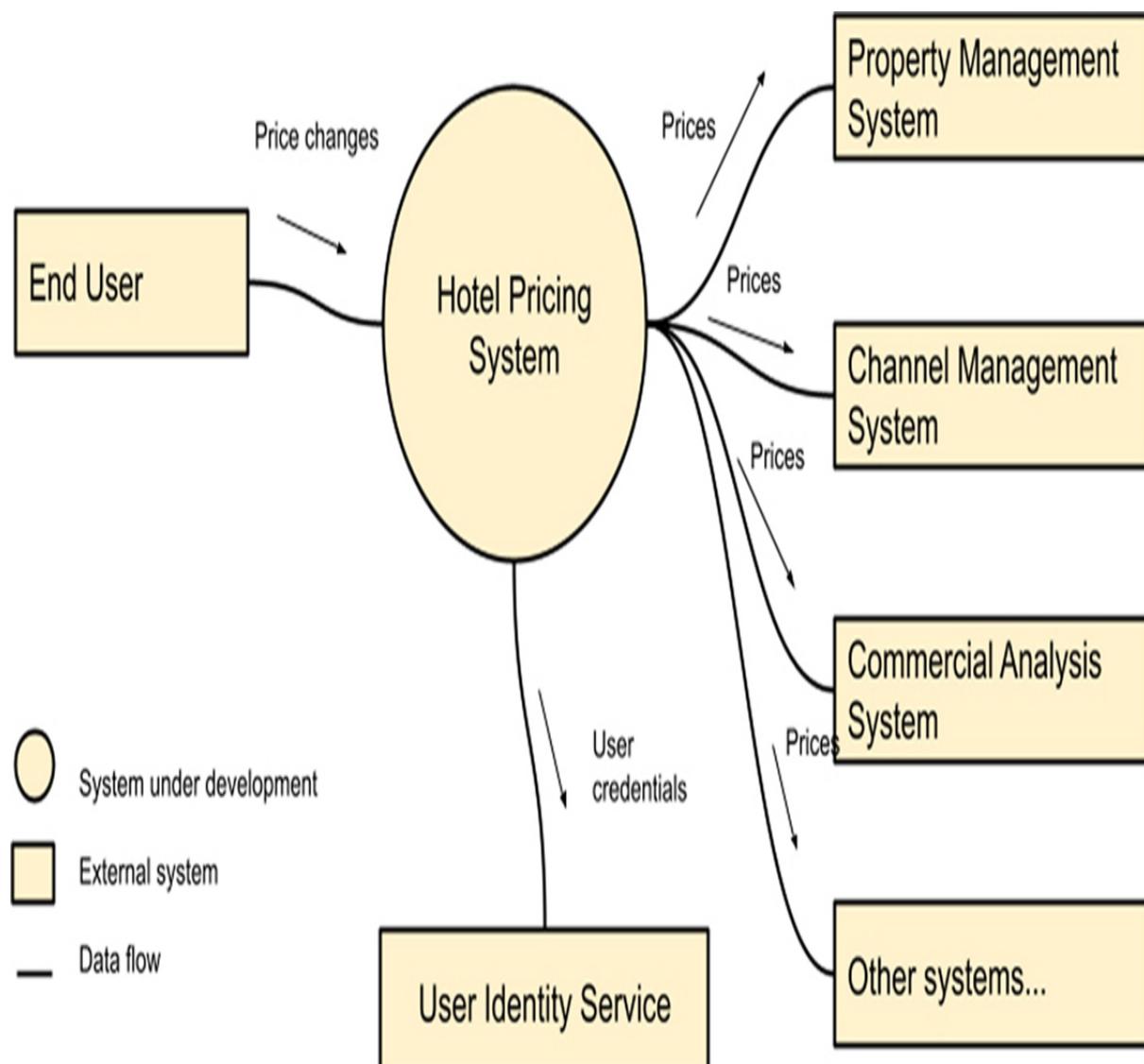


FIGURE 8.1 Context diagram for the Hotel Pricing System

The Hotel Pricing System (HPS) is used by sales managers and commercial representatives to establish prices for rooms at specific dates for the different hotels in the company. Prices are associated with different rates (for example a public rate, or a discount rate) and most of the prices for the different rates are calculated by taking a base rate and applying business rules to it (although some of the rates can also be fixed and not depend on the base rate). Managers typically change prices of the base and fixed rates, and using this information, the Hotel Pricing System calculates the prices of all the rates for all rooms in all hotels which also vary according to the types of rooms available in each hotel. Prices that are calculated by the HPS are used by other systems in the company to make reservations and they are also sent to different online travel agencies through the Channel Management System (CMS). The company's systems are hosted with a cloud provider which offers a user identity service that manages users and provides Single Sign On functionalities.

AD&D Hotels wants to modernize its IT infrastructure, the first step being the complete replacement of the existing pricing system which was developed several years ago and is suffering from reliability, performance, availability and maintainability issues, and this has resulted in financial losses. Furthermore, the company has experienced difficulties because many of its systems are connected using traditional SOAP and REST request-response endpoints: changes to one application frequently impact other applications and complicate the deployment of individual updates to specific applications. Also, the failure of a particular application can propagate through the entire system. Furthermore, some of the applications interact using what are now well-known anti-patterns such as integration through a shared database. Recently revised Enterprise Architecture principles within the organization are

mandating a migration of the system toward a more decoupled model. The remainder of this chapter describes the first three iterations of an architectural design for this system, created using ADD.

8.2 System Requirements

Requirement elicitation activities had been previously performed, and the following is a summary of the most important requirements collected.

8.2.1 Primary functionality

The Hotel Pricing System's functionality is conceptually simple; the main user stories for the system are shown using a Use Case diagram in [Figure 8.2](#).

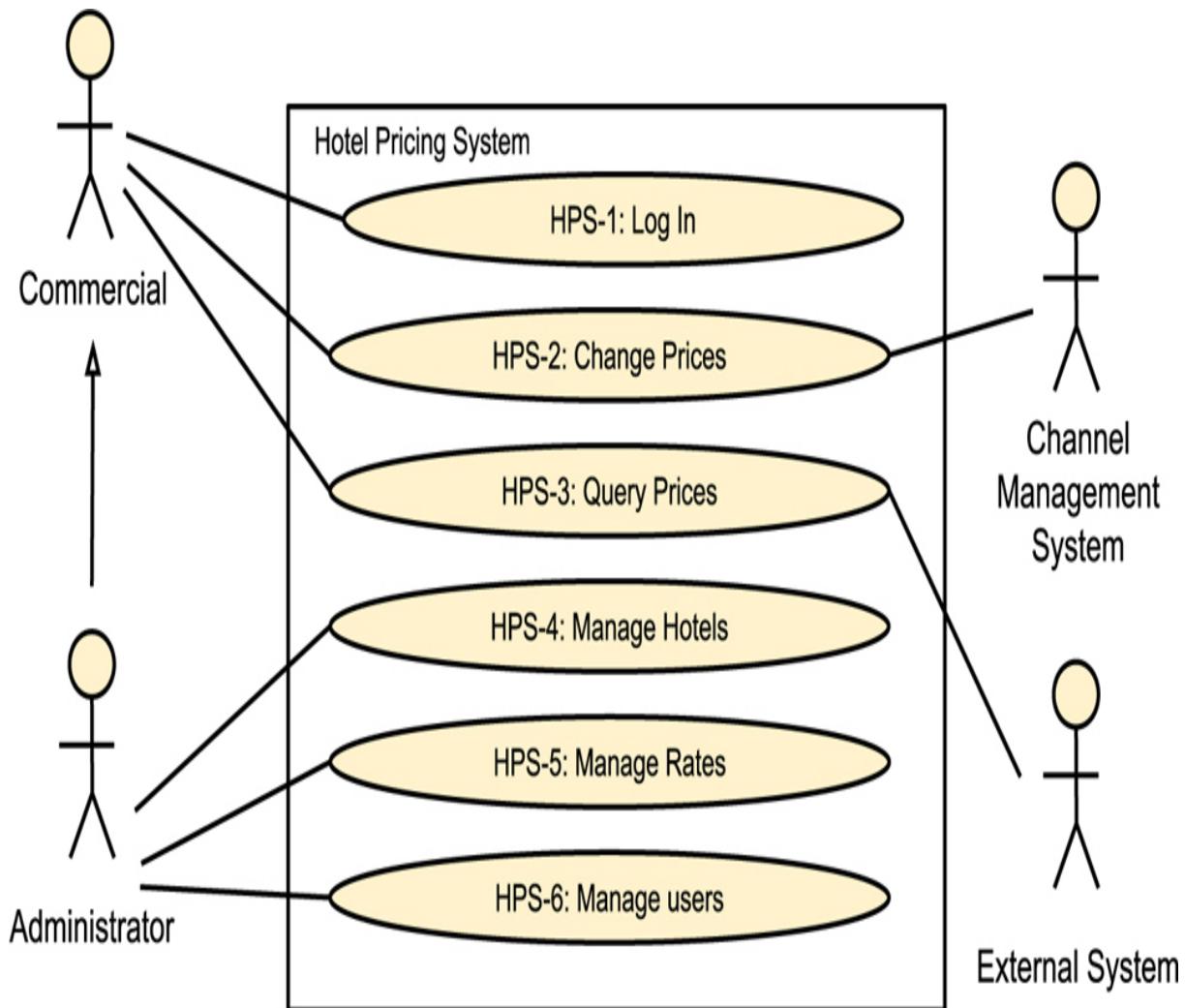


FIGURE 8.2 Initial Use Case Diagram for the Hotel Pricing System

Each of these user cases is described in the following table:

Use case	Description
HPS-1: Log In	A user (commercial or administrator) provides their credentials in a login window. The system checks these credentials against a user identity service and, if successful, provides access to the system. Once logged in, a user can only make queries and changes to the hotels for which they have been authorized.
HPS-2: Change Prices	A user selects a specific hotel for which they are authorized to change prices and selects particular dates where they want to make price changes either to a base rate or a fixed rate. All of the prices for the rates that are calculated from the base rate are calculated at that point. The system allows price changes to be simulated before they are actually changed. When the prices are changed, they are pushed to the Channel Management System and they become available for querying by external systems.
HPS-3: Query Prices	A user or an external system queries prices for a given hotel through the user interface or a query API.
HPS-4: Manage Hotels	An administrator adds, changes or modifies hotel information. This includes editing the hotel's tax rates, available rates, and room types.
HPS-5: Manage Rates	An administrator adds, changes or modifies rates. This includes defining the calculation business rules for the different rates.
HPS-6: Manage Users	An administrator changes permissions for a given user.

8.2.2 Quality attribute scenarios

In addition to the above use cases, a number of quality attribute scenarios were elicited and documented. The seven most relevant and important ones are presented in

the following table. For each scenario we also identify the use case that it is associated with.

ID	Quality Attribute	Scenario	Associated use case
QA-1	Performance	A base rate price is changed for a specific hotel and date during normal operation, the prices for all the rates and room types for the hotel are published (ready for query) in less than 100 ms.	HPS-2
QA-2	Reliability	A user performs multiple price changes on a given hotel. 100% of the price changes are published (available for query) successfully and they are also received by the channel management system.	HPS-2
QA-3	Availability	Pricing queries uptime SLA must be 99.9% outside of maintenance windows.	All
QA-4	Scalability	The system will initially support a minimum of 100,000 price queries per day through its API and should be capable of handling up to 1,000,000 without decreasing average latency by more than 20%.	HPS-3
QA-5	Security	A user logs into the system through the front-end. The credentials of the user are validated against the User Identity Service and, once logged in, they are presented with only the functions that they are authorized to use.	All
QA-6	Modifiability	Support for a price query endpoint with a different protocol than REST (e.g. gRPC) is added to the system. The new endpoint does not require changes to be made to the core components of the system.	All
QA-9	Testability	100% of the system and its elements should support integration testing independently of the external systems	All



8.2.3 Constraints

Finally, a set of constraints on the system and its implementation were collected. These are presented in the following table.

ID	Constraint
CON-1	Users must interact with the system through a web browser in different platforms Windows, OSX, and Linux, and different devices.
CON-4	The initial release of the system must be delivered in 6 months, but an initial version of the system (MVP) must be demonstrated to internal stakeholders in at most 2 months.
CON-5	The system must interact initially with existing systems through REST APIs but may need to later support other protocols.

8.2.4 Architectural concerns

Since this is greenfield development, only a few general concerns were identified initially and these are shown in the following table.

ID	Concern
CRN-1	Establish an overall initial system structure.
CRN-2	Leverage the team's knowledge about Java technologies, the Angular framework and Kafka.
CRN-3	Allocate work to members of the development team.
CRN-4	Avoid introducing technical debt (see chapter 10)

8.3 Development and Operations Requirements

As part of the modernization effort, AD&D hotels also wants to integrate Agile (specifically Scrum) and DevOps practices in the development of the Hotel Pricing System. Besides the previously discussed system requirements, there are also development and operational requirements which need to be taken into account.

Artifacts in the development process move through four different environments which are depicted in [Figure 8.3](#).

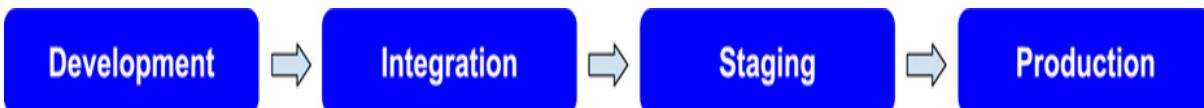


FIGURE 8.3 *Different environments during the development process*

The different environments are the following:

- *Development*: This is a local environment on the developers' computers

- *Integration*: This is an environment in the cloud where an integrated version of the HPS is tested. In this environment, the system is not connected with all of the external systems and so some of these external systems are substituted by mocks.
- *Staging*: This is an environment in the cloud where the system's final tests (including load testing) are performed prior to deployment. Here the system is connected to test versions of all the external systems. At the end of a sprint, the system is typically demonstrated from this environment.
- *Production*: This is the real-world execution environment.

The company is just starting to implement sophisticated deployability practices, so only support for continuous deployment to the integration and staging environments is currently being required. These new requirements, specific to development and operations, result in additional constraints, quality attribute scenarios, and concerns which are presented next.

8.3.1 Quality attribute scenarios

In addition to the quality attribute scenarios that are part of the initial set of system requirements, several scenarios result from development and operations considerations.

ID	Quality Attribute	Scenario	Associated use case
QA-7	Deployability	The application is moved between non production environments as part of the development process. No changes in the code are needed.	All
QA-8	Monitorability	A system operator wishes to measure the performance and reliability of price publication during operation. The system provides a mechanism that allows 100% of these measures to be collected as needed.	HPS-2

8.3.2 Constraints

The additional constraints on the system and its implementation due to considerations of development and operations were collected. These are presented in the following table.

ID	Constraint
CON-2	Manage users through cloud provider identity service and host resources in the cloud.
CON-3	Code must be hosted on a proprietary Git-based platform that is already in use by other projects in the company
CON-6	A cloud-native approach should be favored when designing the system.

8.3.3 Architectural concerns

Development and operations requirements introduce a new concern:

ID	Concern
CRN-5	Set up a continuous deployment infrastructure.

Given these sets of inputs we are now ready to describe the design process, following the steps given in [Chapter 4](#). In this chapter we have only presented the final results of the requirements collection process. The job of collecting these requirements is non-trivial, but beyond the scope of this chapter.

8.4 The Software Design Process

We are now ready to make the leap from the world of requirements and business concerns to the world of design. This is perhaps the most important job for an architect-- translating requirements into design decisions. Of course, many other decisions and duties are important, but this is the core of what it means to be an architect: making design decisions with far-reaching consequences.

8.4.1 ADD step 1 - Review Inputs

The first step of the ADD method involves reviewing the inputs and identifying which requirements will be considered as architectural drivers (which will be included in the design backlog). The inputs are summarized in the following table.

Category	Details																														
Design purpose	<p>This can be considered greenfield development as it involves the complete replacement of an existing system. The purpose of the design activity is to make initial decisions to support the construction of the system from scratch.</p>																														
Primary functional requirements	<p>From the user stories presented in section 8.2.1, the primary ones were determined to be:</p> <ul style="list-style-type: none"> • HPS-2: Change Prices—Because it directly supports the core business • HPS-3: Query Prices—Because it directly supports the core business • HPS-4: Manage Hotels—Because it establishes a basis for many other user stories 																														
Quality attribute scenarios	<p>The scenarios for the HPS have now been prioritized (as discussed in section 2.4.2) as follows:</p> <table border="1" data-bbox="479 819 1393 1812"> <thead> <tr> <th data-bbox="479 819 806 946">Scenario ID</th><th data-bbox="806 819 1067 946">Importance to the customer</th><th data-bbox="1067 819 1393 946">Difficulty of implementation according to the architect</th></tr> </thead> <tbody> <tr> <td data-bbox="479 946 806 1072">QA-1 - Performance</td><td data-bbox="806 946 1067 1072">High</td><td data-bbox="1067 946 1393 1072">High</td></tr> <tr> <td data-bbox="479 1072 806 1199">QA-2 - Reliability</td><td data-bbox="806 1072 1067 1199">High</td><td data-bbox="1067 1072 1393 1199">High</td></tr> <tr> <td data-bbox="479 1199 806 1326">QA-3 - Availability</td><td data-bbox="806 1199 1067 1326">High</td><td data-bbox="1067 1199 1393 1326">High</td></tr> <tr> <td data-bbox="479 1326 806 1453">QA-4 - Scalability</td><td data-bbox="806 1326 1067 1453">High</td><td data-bbox="1067 1326 1393 1453">High</td></tr> <tr> <td data-bbox="479 1453 806 1579">QA-5 - Security</td><td data-bbox="806 1453 1067 1579">High</td><td data-bbox="1067 1453 1393 1579">Medium</td></tr> <tr> <td data-bbox="479 1579 806 1622">QA-6 - Modifiability</td><td data-bbox="806 1579 1067 1622">Medium</td><td data-bbox="1067 1579 1393 1622">Medium</td></tr> <tr> <td data-bbox="479 1622 806 1664">QA-7 - Deployability</td><td data-bbox="806 1622 1067 1664">Medium</td><td data-bbox="1067 1622 1393 1664">Medium</td></tr> <tr> <td data-bbox="479 1664 806 1748">QA-8 - Monitorability</td><td data-bbox="806 1664 1067 1748">Medium</td><td data-bbox="1067 1664 1393 1748">Medium</td></tr> <tr> <td data-bbox="479 1748 806 1812">QA-9 - Testability</td><td data-bbox="806 1748 1067 1812">Medium</td><td data-bbox="1067 1748 1393 1812">Medium</td></tr> </tbody> </table> <p>From this list, QA-1, QA-2, QA-3, QA-4 and QA-5 are selected as primary drivers.</p>	Scenario ID	Importance to the customer	Difficulty of implementation according to the architect	QA-1 - Performance	High	High	QA-2 - Reliability	High	High	QA-3 - Availability	High	High	QA-4 - Scalability	High	High	QA-5 - Security	High	Medium	QA-6 - Modifiability	Medium	Medium	QA-7 - Deployability	Medium	Medium	QA-8 - Monitorability	Medium	Medium	QA-9 - Testability	Medium	Medium
Scenario ID	Importance to the customer	Difficulty of implementation according to the architect																													
QA-1 - Performance	High	High																													
QA-2 - Reliability	High	High																													
QA-3 - Availability	High	High																													
QA-4 - Scalability	High	High																													
QA-5 - Security	High	Medium																													
QA-6 - Modifiability	Medium	Medium																													
QA-7 - Deployability	Medium	Medium																													
QA-8 - Monitorability	Medium	Medium																													
QA-9 - Testability	Medium	Medium																													

Constraints	All of the previously discussed constraints are included as drivers.
Architectural concerns	All of the previously discussed architectural concerns associated with the system are included as drivers.

8.3.2 Iteration 1: Establishing an overall System Structure

This section presents the results of the activities that are performed in each of the steps of ADD in the first iteration of the design process.

8.3.2.1 Step 2. Establish iteration goal by selecting drivers

This is the first iteration in the design of a greenfield system so the iteration goal is to achieve the concern CRN-1 of *establishing an overall system structure*.

Although this first iteration is driven by a general architectural concern, the architect must keep in mind *all* of the drivers that may influence the general structure of the system. In particular the architect must be mindful of the following:

- QA-1: Performance
- QA-2: Reliability
- QA-3: Availability
- QA-5: Security
- CON-1: Access to the system via web browser
- CON-2: Cloud provider
- CON-6: Develop cloud-native solution

- CRN-2: Leverage team's knowledge about Java technologies.
- CRN-4: Avoid introducing technical debt

Making considerations about security is particularly important early in the design process so this quality attribute is given special consideration even though the architect does not consider it to be too difficult to implement.

8.3.2.2 Step 3. Choose one or more elements of the system to refine

Since this involves the complete replacement of an existing system, the first element to refine is the whole HPS system, which is shown in [Figure 8.1](#). In this case, refinement is performed through decomposition.

8.3.2.3 Step 4. Choose one or more design concepts that satisfy the selected drivers.

In this initial iteration, given the goal of structuring the entire system, design concepts are selected according to the roadmap presented in [section 4.3.1](#). The following table summarizes the selection of design decisions. Also, security is considered from the beginning of the design process. The words in **bold** in the following table refer to the design concepts that are selected.

Design decisions and location	Rationale
<p>Structure the back-end part of the system following the CQRS pattern and separate the Command and Query components which communicate through events.</p>	<p>CQRS (Command and Query Responsibility Segregation) is a pattern that separates changes to data from queries to this data. This pattern is useful to support performance, scalability, availability and security, although at the cost of added complexity (see section 7.3.3.2). The command and query components communicate using events. This type of communication using events also supports the decision by the company to move away from connecting applications using traditional request / response invocations. Furthermore, this will allow other applications to connect to the event channel in the future to perform tasks such as analyses or event storage.</p> <p>Discarded alternatives include developing the backend as a monolithic application. The pricing system that is being replaced was structured as a monolith. While this was not the reason for its problems, it is believed that starting with a monolith is not aligned with the enterprise architecture principles of the company and current best practices. Furthermore, if the application is structured as a monolith, scaling it to support future growth will require creating multiple instances of the whole application. While CQRS could be implemented in a monolith, the separation of the application into different components allows the query side of the application to be scaled independently of the command side.</p>
<p>Implement Command and Query components as microservices.</p>	<p>The command and query parts of the system are developed and deployed independently as independent microservices that expose service APIs, making this a service oriented application. Multiple replicas of the query side can be set up to support larger query volumes and higher availability. Security is enhanced because clients accessing the query side cannot make changes to important aspects of the system such as business rule calculations. In addition to the benefits in terms of supporting quality attributes, the use of this pattern in this context is appropriate since all prices do not need to be recalculated with each query and, furthermore, changes in the calculations of business rules do not affect published prices prior to the date of change of the business rule calculation.</p> <p>The use of microservices also allows CRN-3 to be addressed as each microservice can be implemented by a small team.</p>
<p>Secure the APIs exposed by the microservices by implementing the authentication and authorization of actors tactics and also by implementing the ..</p>	<p>The frontend of the system along with external systems will interact with the microservices through APIs. The access to the functionalities provided by the APIs should be secured. Furthermore, access to other resources such as databases needs to be limited since all interactions are made through APIs or the event channel.</p> <p>All of the information that is exchanged through APIs is encrypted.</p>

limit access and encrypt data tactics	
Logically structure the client application using the Rich Internet Application reference architecture.	The system must be accessed from a web browser (CON-1). The frontend part of the system is developed as a Rich Internet Application which consumes APIs on the backend side. Discarded alternatives include developing the application as a rich client or a mobile application.

8.3.2.4 Step 5. Instantiate architectural elements, allocate responsibilities and define interfaces

The instantiation design decisions considered and made are summarized in the following table:

Design Decision and Location	Rationale
Use Apache Kafka as the event bus	<p>Kafka is a durable message broker that enables applications to process, persist and re-process streamed data. Events are published and consumed from “topics” and it supports message ordering through the use of keys, this is necessary since changes to the hotel prices for a given day must be ordered. Kafka also has the benefit of retaining all of the messages in the log. The log acts as the “source of truth” for prices.</p> <p>Discarded alternatives are RabbitMQ and other messaging alternatives which are less familiar to the team (CRN-2).</p>
Create an additional microservice to handle price change events and deliver them to the channel management system	<p>The Channel Management System is a legacy application that cannot be changed to listen to the CQRS events. Instead, an additional microservice is created so that when it receives events from the Command side, these are pushed to the Channel Management System.</p> <p>This microservice will not require a database, since messages are pulled from the log (Kafka) and sent to the CMS system. In case the CMS is not available, messages are put back in the log and delivery is retried.</p> <p>In the future, if the CMS system is updated or replaced, its integration with the pricing system will require no changes to the HPS.</p>
Use Angular to implement the frontend component.	<p>Angular is selected because the team has experience with this particular framework and because it supports the development of responsive frontends. This is necessary to support CON-1.</p> <p>Discarded alternatives are other web frameworks such as React or Vue.js, mainly because the team is not familiar with them.</p>
Use Docker containers to promote portability across environments.	<p>As the application has to be moved across environments (QA-7), deploying the application as a set of container images is preferred. Each microservice is deployed in a container. This ensures that the application will run in the same execution environment from development to production.</p> <p>A discarded alternative is the use of virtual machines as the size of VM images is typically considerably larger than container images; also VMs require more resources (memory, CPU) than containers.</p>
Use TLS (HTTPS) to encrypt data in transit.	<p>All of the data in transit, that is the one that is exchanged with the frontend and external systems is encrypted. At this point no decisions have been made regarding data at rest (in the databases).</p>
Secure the API	<p>Access tokens are a proven authentication and authorization mechanism for APIs. These</p>

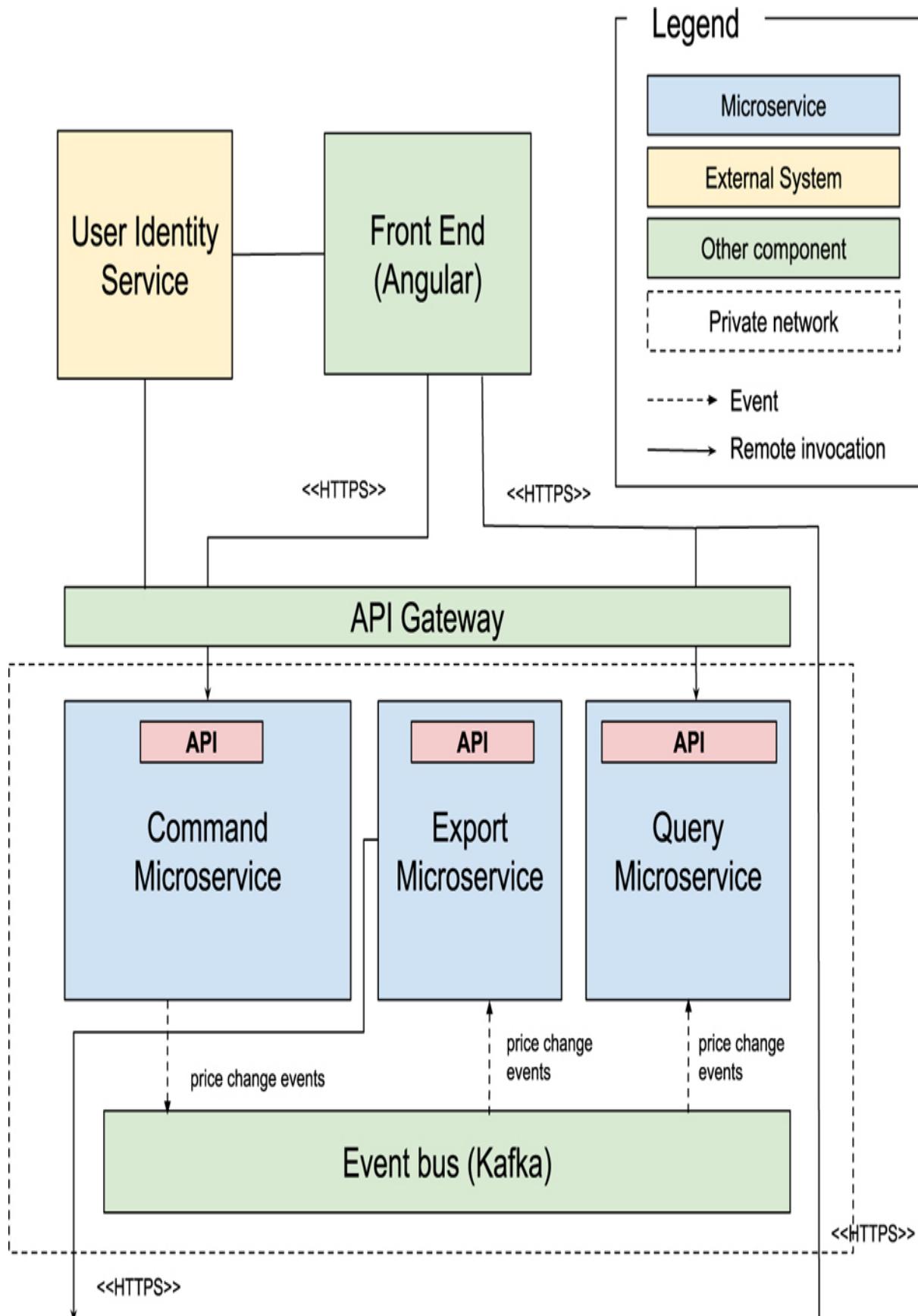
through the use of JSON Web Tokens	tokens are provided by the User identity Service once a user has successfully provided their credentials.
Use an API Gateway to limit access to the microservices that expose the APIs	Access to the microservices is not performed directly but rather through an API gateway which can perform different functions including securing and monitoring the APIs. The calls that are received by the API are routed to the microservices which are protected inside a private network in the cloud.

The results of these instantiation decisions are recorded in the next step.

In this initial iteration it is typically too early to precisely define functionality and interfaces. In the next iteration, which is dedicated to defining functionality in more detail, interfaces will begin to be defined.

8.3.2.5 Step 6. Sketch views and record design decisions

Figure 8.4 shows the sketch of a component view of the architecture adapted according to the design decisions we have made.



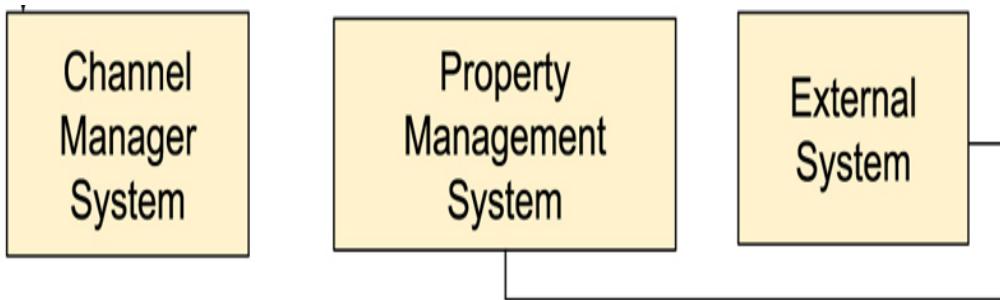


FIGURE 8.4 *Main components of the Hotel Pricing System*

This sketch is created using an informal representation. A short description of the responsibilities of the elements is also captured. The fact that each component has a specific function promotes high cohesion and thus helps avoid technical debt (CRN-4). Note that the descriptions at this point are quite crude, just indicating major functional responsibilities, with no details. The following table summarizes the information that is captured:

Element	Responsibility
Front-end	Represents the client-side application built using the Rich Internet application reference architecture and implemented using Angular
API Gateway	The API Gateway sits between the clients of the API and the microservices that expose them.
Command Microservice	This microservice encapsulates the logic for the Command part of the CQRS pattern. This includes managing the different hotels, rates, room types and the price calculation business rules.
Query Microservice	This microservice's primary responsibility is to serve price requests. It exposes two APIs, one which is used by the front-end and another which is used by legacy internal systems.
Export Microservice	This microservice's primary responsibility is to consume price change events and publish them on the Channel Management System.
Event Bus	This is the event bus that is used by the Command microservice to publish price change events, and by the Query and Export microservices to consume these events.
Property Management System	This element represents the legacy Property Management System which needs to query prices.
Channel Management System	This element represents the legacy channel management system which must be notified of changes in the prices. The export microservice is in charge of notifying this system.
External System	This element represents any other system that needs to query prices.
User Identity Service	Service from the cloud provider which manages users and provides access tokens for the APIs.



In this iteration, initial decisions about how to deploy the application are made. The microservices and the event bus are deployed inside a private network and are only accessible through the API gateway. Decisions about specific cloud resources to be used still need to be made.

8.3.2.6 Step 7. Perform analysis of current design and review iteration goal and achievement of design purpose

The decisions made during this iteration result in new concerns that need to be addressed:

- CRN-6 Select specific cloud technologies

The following table summarizes the design progress using the Kanban technique discussed in 4.8.2.

Not Addressed	Partially Addressed	Completely Addressed	Design decisions made during the iteration
	HPS-1		Login will be performed using the cloud provider identity management service but additional decisions need to be made.
	HPS-2		The overall structure of the application was decided based on the need to support this use case, but many details have not yet been addressed.
	HPS-3		The use of CQRS and the creation of a specific microservice to support queries was made to support this use case but many details have not yet been addressed.
HPS-4			No relevant decisions made yet.

HPS-5			No relevant decisions made yet.
HPS-6			No relevant decisions made yet.
	QA-1		Separation of command and query sides will promote performance but many details have not yet been addressed.
	QA-2		The selection of Kafka, a highly reliable message broker supports this scenario, but there are many decisions such as the configuration of Kafka itself that have not yet been addressed.
	QA-3		The use of CQRS, Kafka, microservices and a container management service in the cloud provider support this scenario, but many details have not yet been addressed.
	QA-4		Scalability on the query side will be supported by creating multiple replicas of the query microservice. Additional decisions to make this microservice scalable need to be made.
	QA-5		Use of an API gateway, HTTPS and access tokens contribute towards the satisfaction of this driver.
QA-6			No relevant decisions made yet.
	QA-7		The use of container technologies is a starting point to support this driver but additional decisions need to be made.
QA-8			No relevant decisions made yet.
	CON-1		The front-end will run in a web browser; the Angular framework is well supported by modern browsers.
	CON-2		While this constraint has been considered, a new concern has appeared (CRN-6) regarding the selection and configuration of products from the cloud provider.
CON-3			No relevant decisions made yet.
	CON-4		Initial decisions, such as the identification of microservices, which can

			help estimate development time have been made but additional information is needed.
CON-6			No decisions about specific cloud provider resources have been made yet.
	CRN-1		Establishing an overall initial system structure was the goal of this iteration.
	CRN-2		Decisions regarding the implementation of the microservices have not yet been made, but Angular has been chosen as the implementation framework for the front-end.
	CRN-3		The decomposition of the system into different microservices will allow work to be allocated to members of the development team (one small team per microservice).
	CRN-4		Microservices have specific functions, promoting high cohesion and modifiability.
CRN-5			No relevant decisions made yet.
CRN-6			No relevant decisions made yet.

8.3.3 Iteration 2: Identifying structures to support primary functionality

This section presents the results of the activities that are performed in each of the steps of ADD in the *second* iteration of the design process for the Hotel Pricing System. In this iteration we move from the relatively abstract view of the architecture developed in iteration 1 to more detailed decisions that will drive implementation.

This movement from the generic to the specific is intentional, and is built into the ADD method. We can't design everything up front, so we need to be disciplined about what decisions we make, and when, to ensure that the design is done in a systematic way, addressing the biggest risks first and moving from there to ever finer details. Our goal for the first iteration was to establish an overall system structure. Now that this goal has been met, our goal for this second iteration is to reason about the units of implementation, which affect team formation, interfaces, and how development tasks may be distributed, outsourced, and implemented in sprints.

8.3.3.1 Step 2. Establish iteration goal by selecting drivers

Iteration goal: The goal of this iteration is that of *identifying structures to support primary functionality*. Identifying these elements is useful for understanding how functionality is supported and is also helpful in providing a better estimate for development:

In this second iteration, the architect considers the system's primary user stories:

- HPS-2: Change prices
- HPS-3: Query prices

It should be noted that in this iteration, the design is focused on the back-end part of the application.

8.3.3.2 Step 3. Choose one or more elements of the system to refine

The elements that will be refined in this iteration are the command, query and export microservices derived from the CQRS pattern used in the first iteration.

8.3.3.3 Step 4. Choose one or more design concepts that satisfy the selected drivers.

The following table summarizes the design decisions made. The words in **bold** refer to the design concepts that are selected.

Design decisions and location	Rationale and assumptions
Create a Domain Model for the application.	<p>Before starting a functional decomposition it is necessary to create an initial domain model for the system, identifying the major entities in the domain, and their relationships. While this is not <i>per se</i> an architectural decision, it is important and interacts with all subsequent architectural choices</p> <p>There are no shortcuts or good alternatives for this step. A domain model must eventually be created, or it will emerge in a sub-optimal fashion leading to an ad hoc architecture that is hard to understand and maintain.</p>
Logically structure the microservices using the Service Application reference architecture. Microservices expose APIs.	<p>Service Applications (see 3.2.1) do not provide a user interface but rather expose services which are consumed by other applications.</p> <p>This reference architecture is suitable to logically structure each microservice that exposes an API. These APIs will be consumed by the client application and by other legacy systems (CON-5).</p> <p>There are no discarded alternatives with respect to the reference architecture.</p>

8.3.3.4 Step 5. Instantiate architectural elements, allocate responsibilities and define interfaces

The instantiation design decisions made in this iteration are summarized in the following table:

Design Decision and Location	Rationale
Create a domain model for the command side microservice.	<p>The domain model that is to be created is not a domain model for the entire company, but rather a model that is useful in the context of hotel prices. This model resides on the command side microservice.</p> <p>The query and export side microservices do not have a specific domain model as there was not a compelling reason to create them. In the case of the query side, price change events are stored directly in the data store. In the case of the export side, no information is stored as we will see later.</p>
Create a PriceChangeEvent and use a single topic and Kafka keys to ensure temporal ordering of price changes.	<p>Kafka has been chosen in the previous iteration, and in this iteration further decisions are made:</p> <ul style="list-style-type: none"> • A single topic is used for price change events; these events contain the prices for a given hotel and date. • A key that guarantees price ordering is defined. Furthermore, this key allows log compaction to be supported.
Use a relational database for the Command microservice	<p>Given that the command microservice manages the entities of the domain model (Hotel prices, Rates, Room Types, etc.) and given that these entities are related to each other, the most natural and appropriate mechanism for storing them is a relational database.</p> <p>Discarded alternatives are non-relational databases or other types of storage.</p>
Use a non relational database for the Query Microservice	<p>Since the database on the query side does not need to handle transactions and relationships between entities and since the prices that are stored have a variable structure depending on the hotel, it was decided that the use of a non-relational database is more appropriate.</p> <p>PriceChangeEvent can be appropriately stored in a document database.</p> <p>Discarded alternatives include other types of NoSQL, relational databases, or other types of storage.</p>
Use the Spring	The Spring Framework technology stack is selected because the team is familiar with this

Framework technology stack to implement the microservices and map the system user stories to service application reference architecture modules.

framework. Since the team is familiar with the Spring Framework technologies (CRN-2), no alternative frameworks for languages other than Java are considered.

The service application reference architecture and Spring framework provide categories of modules for the different layers: controllers to expose APIs, services to manage business

logic, entities to manage business entities and repositories to persist them, among others.

User stories can typically be mapped to modules following well-established rules: one entity can be mapped to at least one controller, one service and one repository. This technique ensures that modules that support all of the functionalities and entities are identified.

	<p>The architect will only perform this task for the primary use cases. This allows another team member to identify the rest of the modules to allocate work among team members (CRN-3).</p> <p>Having established the set of modules, the architect realizes the need to test these modules and so a new architectural concern was identified here:</p> <ul style="list-style-type: none"> ▪ CRN-7: Ensure that all non-generated modules can be unit tested <p>The reason that only “non-generated” modules are covered by this concern is that some modules such as repositories are generated by Spring and do not need to be unit tested. The use of Spring facilitates testability since it is built upon the Inversion of Control (IoC) pattern.</p>
Define and expose REST APIs to the front-end and document them using Swagger/OpenAPI	<p>The API exposed by the microservices for consumption by the front-end will follow the REST paradigm and exchange JSON data. REST is selected because it is commonly used to communicate with Rich Internet front-ends and in this case there are no requirements that would constrain the use of this paradigm. Furthermore, JSON is selected as the data exchange format for similar reasons.</p> <p>A new concern arises from this decision:</p> <ul style="list-style-type: none"> ▪ CRN-8: Document the API <p>This new concern is immediately addressed by selecting Swagger/OpenAPI as the framework used to document the API. Swagger is selected because it is a mature framework which integrates easily with the Spring Framework.</p>
Apply the Externalize configurations tactic to promote portability.	<p>To support moving from one execution environment to another without the need of changing code or repackaging, configuration is externalized and environment variables are used to configure aspects such as database connections.</p>
Use cloud provider managed services for databases, container orchestration and Kafka.	<p>After evaluating the alternative products available from the cloud provider considering aspects such as cost, licensing, ease of installation, and administration, a decision was made to select products directly managed by the cloud provider for the databases, container orchestration and Kafka.</p> <p>There are no anticipated needs to port this application to a different cloud provider once <small>deployed on open source solutions that can be deployed on the cloud but that require more</small></p>

management are discarded. While these solutions could be cheaper than managed services, the time required to set them up is considerable and economic analysis of the costs of managed services convinced management that they are acceptable (CON-6).

Although the products are now selected, configuring them adequately is still pending. A new concern is raised:

- CRN-9: Configure managed cloud services

Please note that while the structures and interfaces are identified in this step of the method they are only captured in the next step. For this reason they are not shown here.

8.3.3.5 Step 6. Sketch views and record design decisions

[Figure 8.5](#) shows an initial Domain Model to be used in the command side microservice:

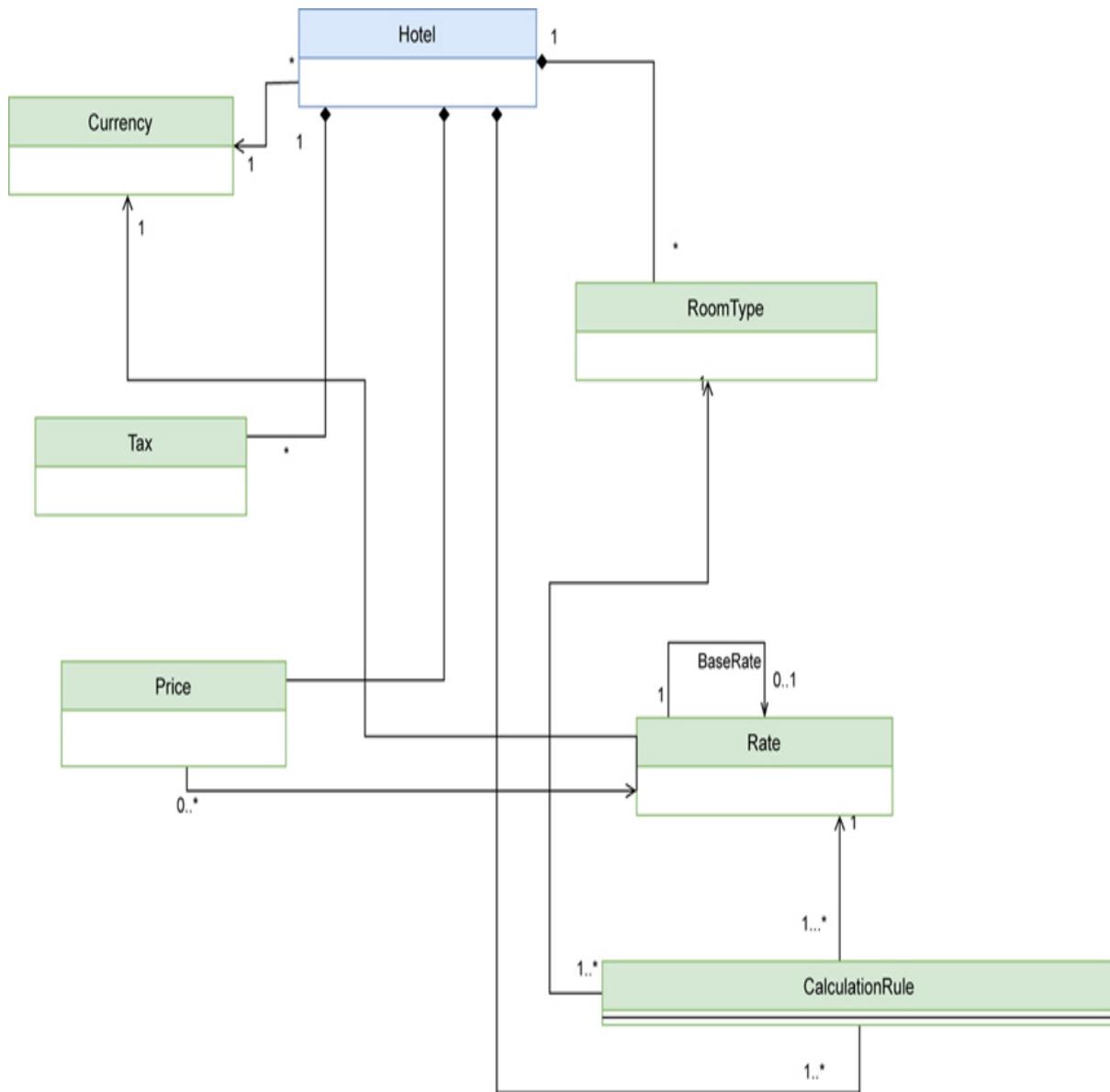


FIGURE 8.5 Domain model for the command side of the Hotel Pricing System (Key: UML)

The entities of the domain model are the following:

Element	Responsibility
Hotel	A hotel, which is the main entity.
Currency	The primary currency associated with the hotel.
RoomType	A particular type of room. One hotel may have different types of rooms.
Tax	Taxes associated with the hotels.
Rate	Different room rates that are used to calculate prices.
CalculationRule	Rules that are applied to rates to calculate prices.
Price	The price in the base rate (or a fixed rate) for a particular date.

Figure 8.6 diagram shows a sketch of a module view of the command side microservice with modules derived from the entities which are associated with the primary use case (HPS-2).

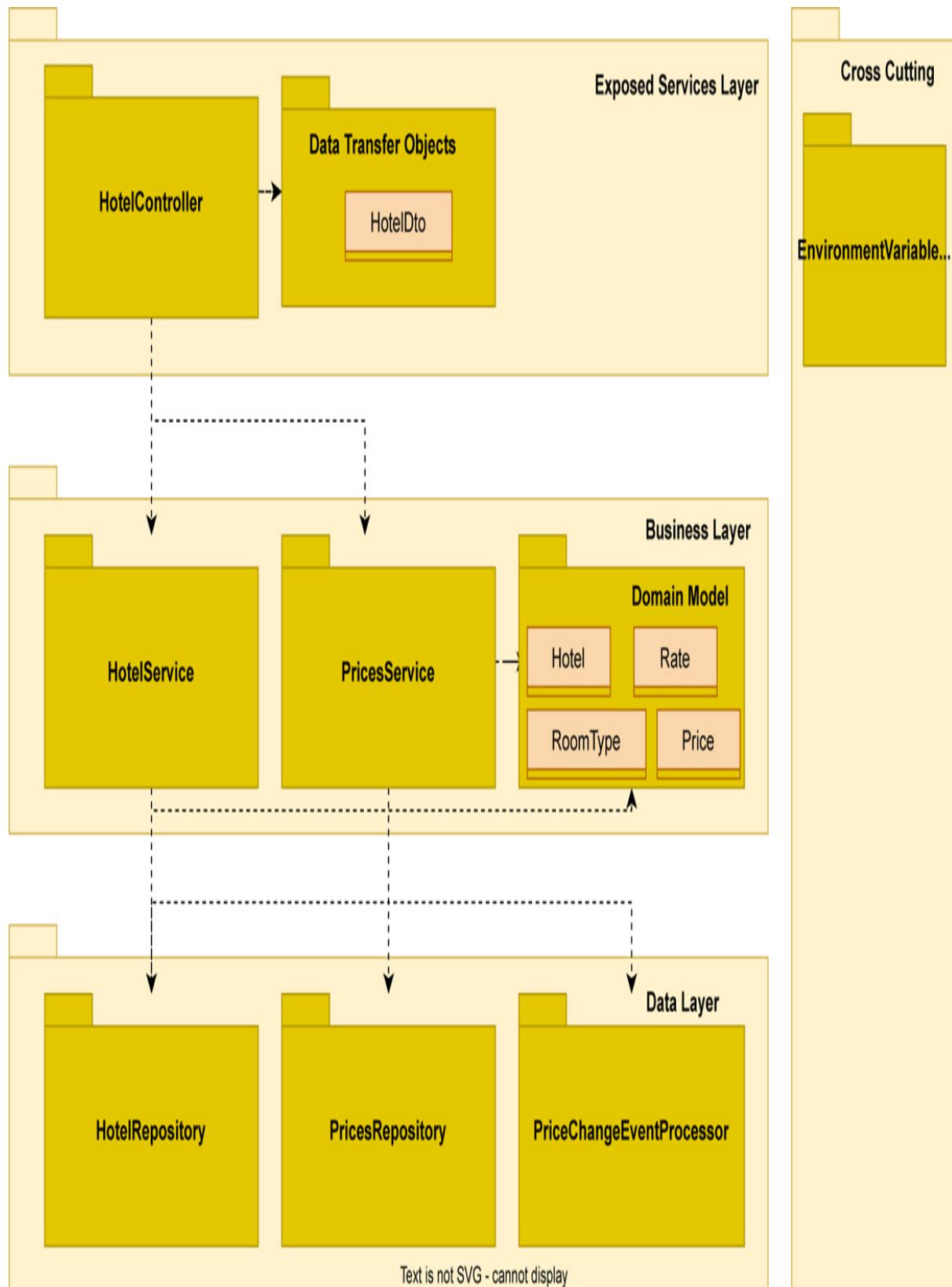


FIGURE 8.6 *Module view of the command side microservice
(Key: UML)*

The responsibilities for the elements identified in the previous diagram are the following:

Element	Responsibility
HotelController	Exposes API endpoint for CRUD operations for the Hotel resource and contains annotations that produce Swagger documentation.
DataTransferObjects	Contains objects that are returned or received by the API endpoint.
HotelService	Validates certain business rules that do not fit inside entities and coordinates objects from the domain model and data layer to support operations exposed by the controller. Also manages certain transactional operations.
PricesService	This element coordinates price changes.
Domain Model	Contains the business entities. These entities are “rich” in the sense that they contain business logic.
HotelRepository	Manages persistence of Hotel objects.
PricesRepository	Stores public rate prices for given days. Prices for other rates are not stored since they are calculated and sent as events to the query side.
PriceChangeEventProcessor	Generates events related to price changes.
EnvironmentalVariableManager	Manages externalized configuration values (environment variables).

[Figure 8.7](#) shows a sketch of a module view of the query side microservice.

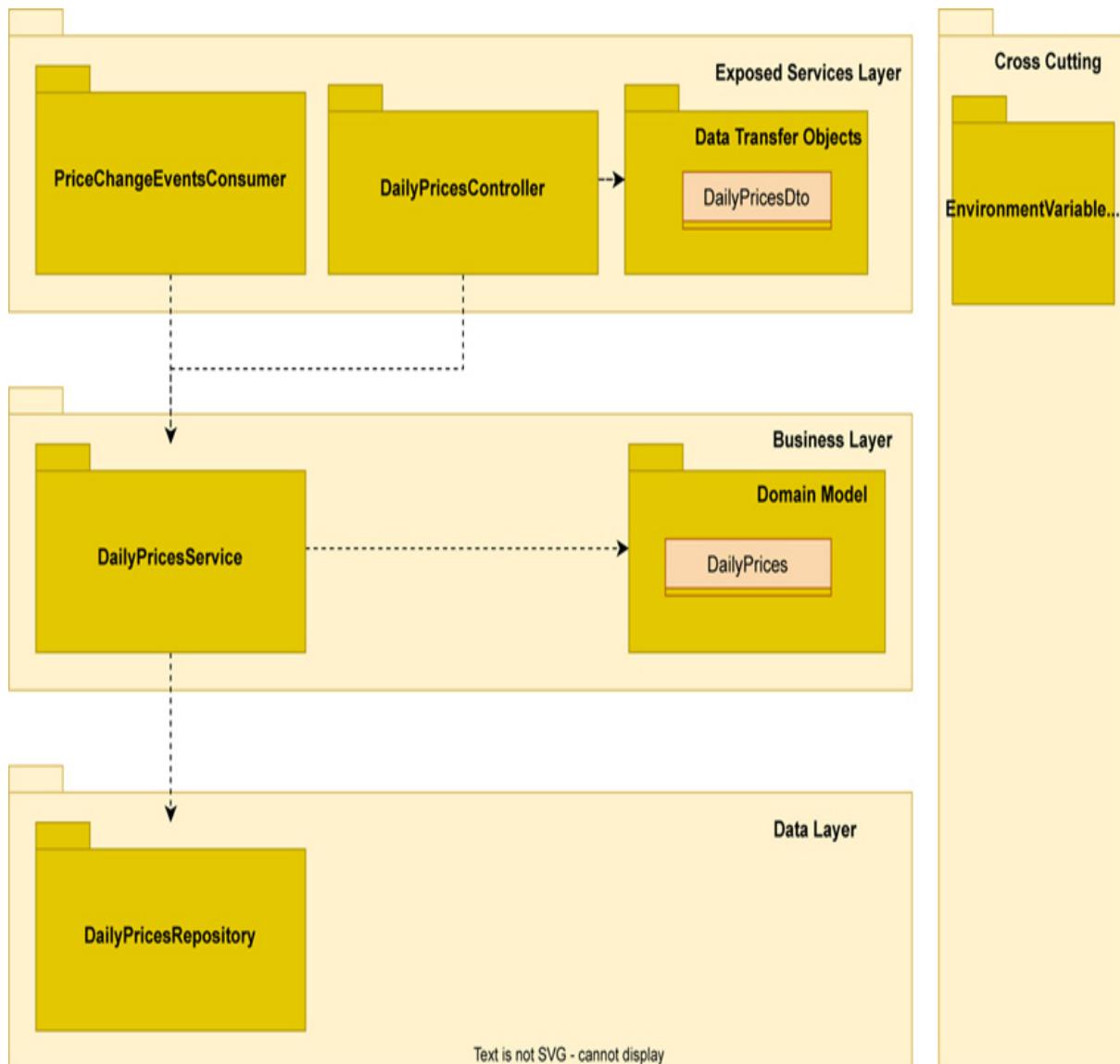


FIGURE 8.7 Module view of the query side microservice
(Key: UML)

The responsibilities of the elements identified in the previous diagram are the following:

Element	Responsibility
PriceChangeEventsConsumer	Receives price change events that are published in Kafka
DailyPricesController	Exposes APIs to query prices
DailyPricesService	Prepares price change events to be stored
DailyPricesRepository	Stores events in the (non-relational) database
Data Transfer Objects	Prices that are returned as JSON or XML objects
Domain Model	Contains a class that stores the information from the price change events
EnvironmentalVariableManager	Manages externalized configuration values (environment variables).

Figure 8.8 shows a sketch of a module view of the export side microservice.

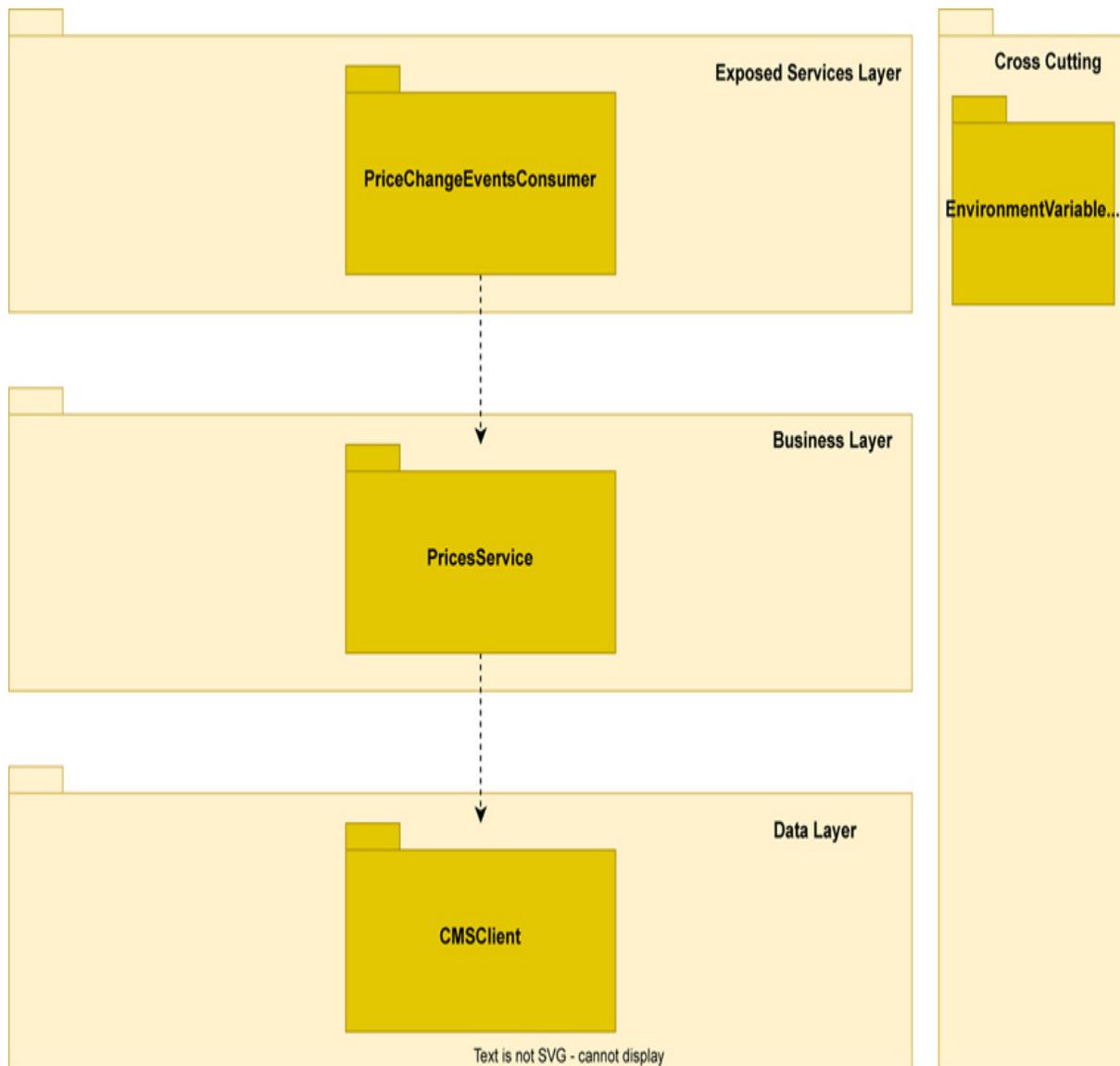


FIGURE 8.8 Module view of the export side microservice
(Key: UML)

The responsibilities of the elements identified in the previous diagram are the following:

Element	Responsibility
PriceChangeEventsConsumer	Receives price change events that are published in Kafka.
DailyPricesService	Prepares payload to be sent to the CMS.
CMSClient	Communicates with the CMS.
EnvironmentalVariableManager	Manages externalized configuration values (environment variables).

Below we show the sequence diagrams for HPS-2, which are used to define interfaces (as discussed in [section 4.6](#)).

HPS-2 (Change prices)

Command Side:

[Figure 8.9](#) shows an initial sequence diagram for HPS-2 (Change Prices) on the command side. The diagram shows how the client application invokes the POST method that triggers a change in prices. The PriceService retrieves the hotel and price that must be changed and asks the hotel object to calculate the additional types of prices. To increase performance, an index is added to the database. Once prices are calculated, an event is created and sent to the PriceChangeEventProcessor which sends it to Kafka.

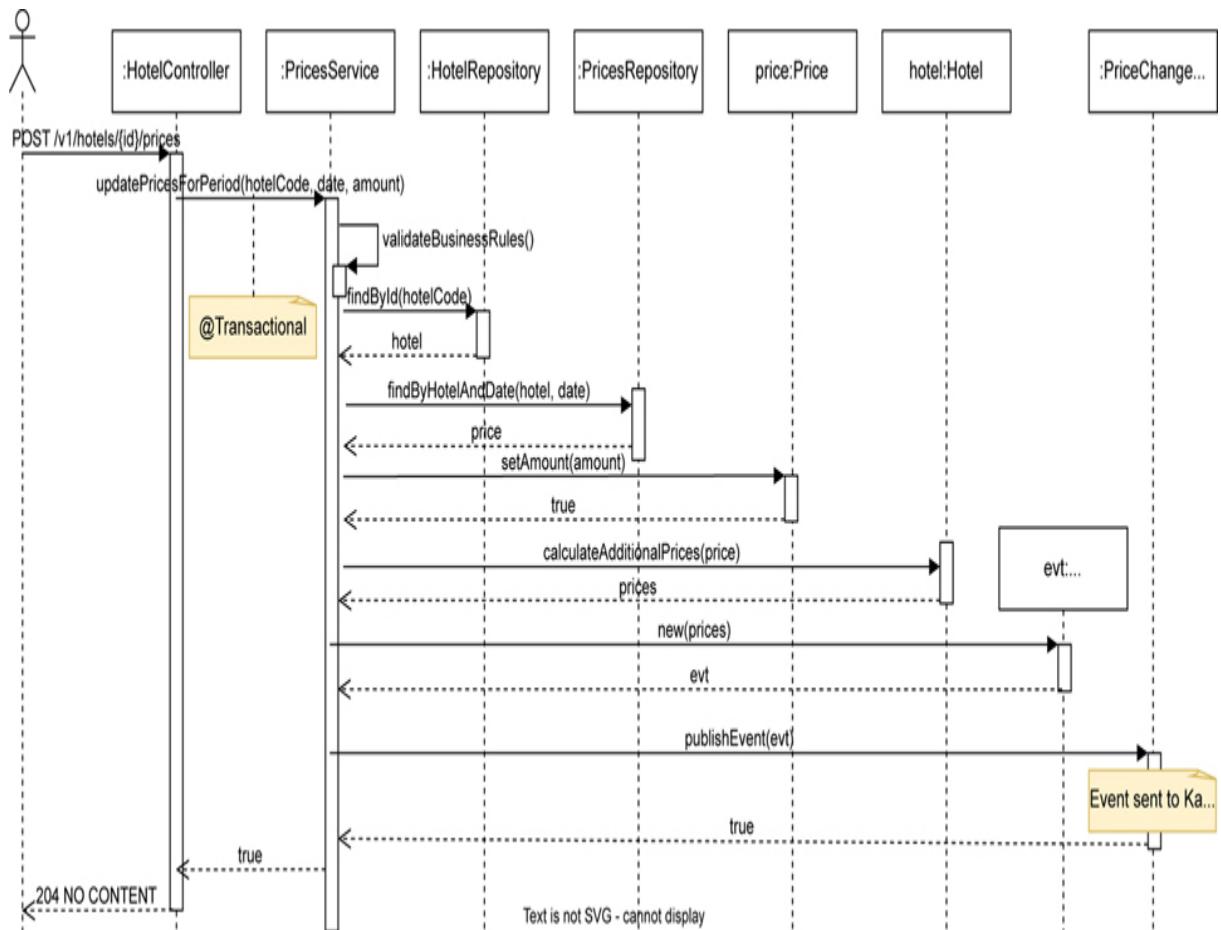


FIGURE 8.9 Sequence diagram for HPS-2 (Change Prices) on the command side(Key: UML)

It should be noted that the `PriceChangeEventProcessor` must use an adequate topic and keys to send the message to ensure ordering of the events. And the schema for the event payload remains to be defined. Also, it is necessary to ensure that a failure in the broker does not result in a price that is modified but an event not sent (the method is annotated as `@Transactional`, but configuring a transaction manager is pending). Finally the event broker must also be configured to support adequate performance and reliability. All of these aspects can be considered as part of the additional concern that is created in this iteration: CRN-9: configure managed cloud services.

Query side:

Figure 8.10 shows the interaction that occurs on the query side when the event is received. Here the event payload is simply stored in the database.

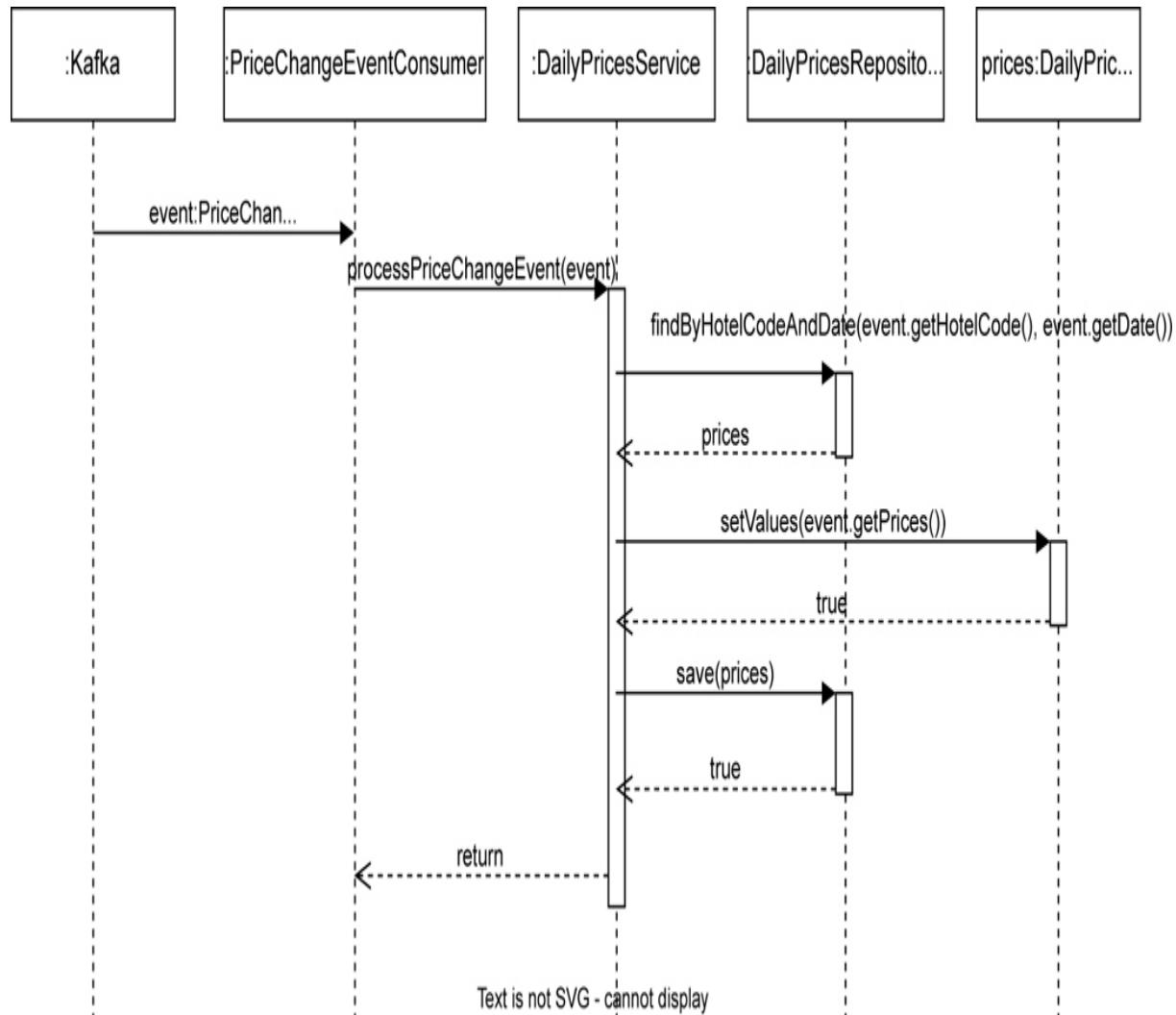


FIGURE 8.10 Sequence diagram illustrating the reception of events in the query side (Key: UML)

Export side:

Figure 8.11 shows the interaction that occurs on the export side when the event is received. In this scenario, the event is consumed and the event contents are used to prepare the

payload that is sent to the channel management system. After the event is sent, a success message is returned to Kafka.

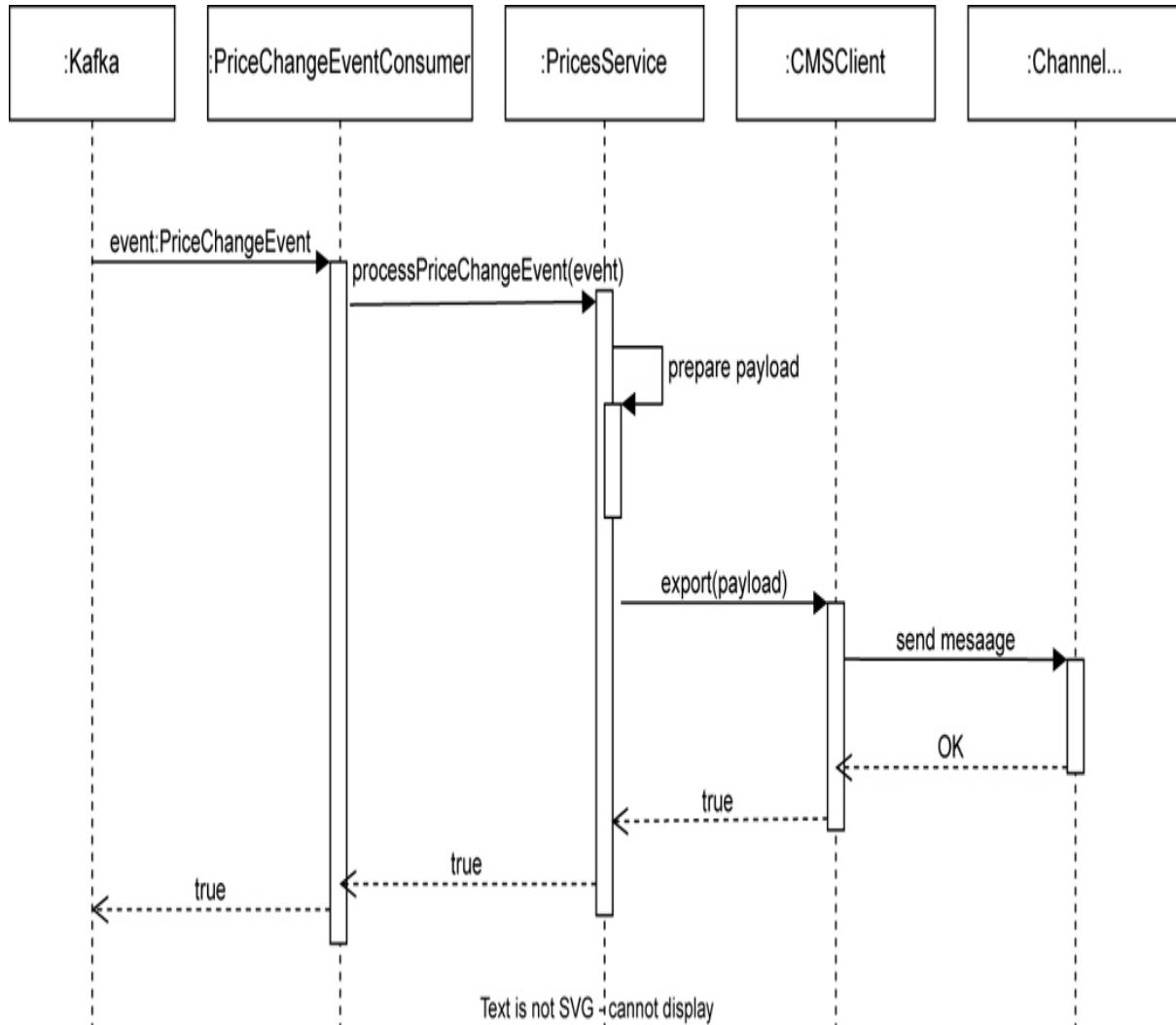


FIGURE 8.11 Sequence diagram illustrating the reception of events in the export side (Key: UML)

From the interactions identified in the sequence diagrams, initial interfaces can be identified. In this case, both an external interface (API) and internal interfaces are identified. Regarding the API, endpoint paths are defined from the domain model. It should also be noted that API versioning (an additional concern) is considered in the name of the path. A more detailed description of the methods will

eventually need to be provided but here we only provide a summary of the most relevant methods.

HotelController:

Method name	Description
POST /v1/hotels/{id}/prices	Allows prices for a hotel to be updated; returns HTTP NO CONTENT if change is successful, or HTTP BAD REQUEST if parameters are incorrect. Parameters: <ul style="list-style-type: none">▪ Hotel Identifier▪ Range of dates of price changes▪ Amount

The following are interfaces from the modules, and they provide a basis for defining internal interfaces. Only a sample of methods from the command side are described:

PriceServices:

Method name	Description
updatePriceForPeriod	<p>Update prices for a hotel and produce a price change event.</p> <p>Parameters:</p> <ul style="list-style-type: none"> ▪ Hotel Identifier ▪ Range of dates of price changes ▪ Amount <p>Returns:</p> <ul style="list-style-type: none"> ▪ boolean: true if successful, false otherwise <p>Throws:</p> <ul style="list-style-type: none"> ▪ PriceChangeEventException in case price change fails.

PriceRepository:

Method name	Description
findByHotelAndDate	<p>Retrieves a price object associated with a hotel and a particular date.</p> <p>Parameters:</p> <ul style="list-style-type: none"> ▪ hotel object ▪ date <p>Returns:</p> <ul style="list-style-type: none"> ▪ price object or null

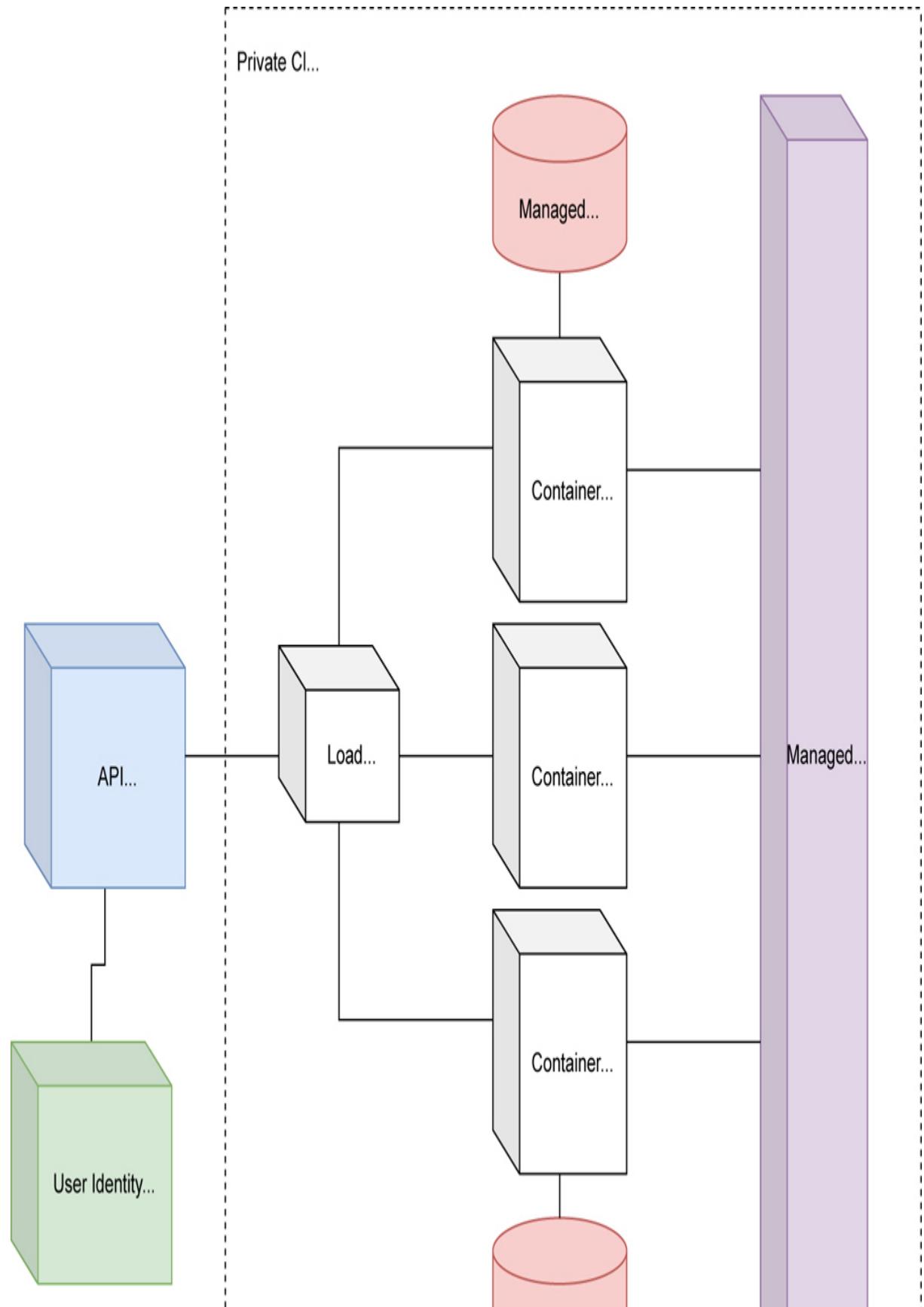
Hotel:

Method name	Description
calculateAdditionalPrice s	<p>Calculates prices associated with all the rates and room types for this hotel. Returns a data structure with the prices.</p> <p>Parameters:</p> <ul style="list-style-type: none"> ▪ price for base room <p>Returns:</p> <ul style="list-style-type: none"> ▪ prices for other rooms and rates

PriceChangeEventProcessor:

Method name	Description
publishEvent	<p>Publishes an event to the broker for the appropriate topic with a key that ensures correct ordering.</p> <p>Parameters:</p> <ul style="list-style-type: none"> ▪ PriceChangeEvent <p>Returns:</p> <ul style="list-style-type: none"> ▪ prices for other rooms and rates

Regarding deployment, the design decisions have resulted in a clearer vision of the solution in the cloud environment. At this point specific products from the cloud provider have been selected but they still need to be configured appropriately to support the quality attribute goals (CRN-9). [Figure 8.12](#) presents an allocation view of the system using cloud-managed resources.



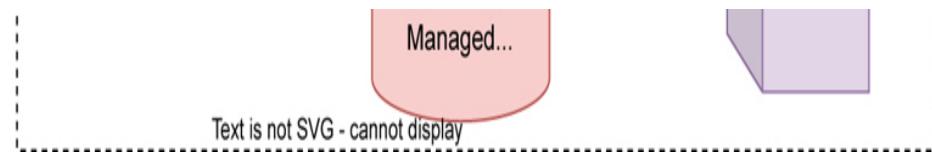


FIGURE 8.12 Initial deployment diagram of the system
(Key: UML)

The responsibilities of the elements identified in the previous diagram are the following:

Element	Responsibility
User Identity Management Service	Manages users and provides and validates access tokens used to access APIs.
API Gateway	Restricts direct access to the microservices and secure the APIs. Also responsible for other aspects such as monitoring.
Load Balancer	Fowards and balances the requests to the microservices inside the private cloud network.
Container Orchestration Service	Provides an execution environment for the containers associated with the different microservices. This service also monitors the health of the containers and, if needed, restarts them in case of failures, contributing to improved availability (QA-3).
Managed Relational Database	This is a database service managed by the cloud provider.
Managed Non relational Database	This is a database service managed by the cloud provider.
Managed Kafka Service	This is a Kafka service managed by the cloud provider.

8.3.3.6 Step 7. Perform analysis of current design and review iteration goal and achievement of design purpose

The decisions made in this iteration provided an initial understanding on how functionality is supported in the system. Furthermore, the design decisions made at this point are sufficient to produce a Minimum Viable Product for the system that can be demonstrated to internal stakeholders and that supports price changes and queries to these changes (CON-4).

New architectural concerns are added as a result of the decisions made in this iteration:

- *CRN-7*: Ensure that all non generated modules can be unit tested
- *CRN-8*: Document the APIs
- *CRN-9*: Configure cloud provider managed services

The following table summarizes the design progress using the Kanban technique discussed in 4.8.2. Note that drivers that were completely addressed in the previous iteration are removed from the table.

Not Addressed	Partially Addressed	Completely Addressed	Design decisions made during the iteration
	HPS-1		No additional decisions were made during the iteration
	HPS-2		This iteration was dedicated to the identification of the modules that support this use case on the back-end, but the front-end part is not yet designed.
		HPS-3	Querying through the API has been addressed during this iteration
	HPS-4		No relevant decisions have been made yet but rules for module identification used for HPS-2 can also be applied for this use case.
	HPS-5		idem
	HPS-6		idem
	QA-1		Adding an index to the database will accelerate the retrieval of the information required by HPS-2 but additional decisions need to be made.
	QA-2		Kafka is highly reliable and the use of a transaction manager to ensure that an event is sent when prices are changed will promote reliability.
	QA-3		No additional decisions were made during this iteration.
	QA-4		No additional decisions were made during this iteration.
	QA-5		Use of an API gateway, HTTPS and access tokens contribute towards the satisfaction of this driver.



		QA-6	The way the query microservice is structured, based on the service application reference architecture, will help add other types of query endpoints.
	QA-7		The use of containers and externalized configurations (through EnvironmentalVariableManager) should allow this requirement to be satisfied. There are still some decisions to be made regarding the way configurations will be externalized and how containers will be built.
QA-8			No relevant decisions made yet.
CON-3			No relevant decisions made yet.
	CON-4		The level of decomposition achieved in this iteration provides sufficient information to estimate and understand if the time constraint can be supported. Also the result of this iteration can already support the primary functionality and can be demoed.
	CON-6		Cloud managed resources have been selected.
	CRN-2		The Spring framework was selected to implement the microservices.
	CRN-4		The decisions made in this iteration promote high cohesion and modifiability.
CRN-5			No relevant decisions made yet.
	CRN-6		Specific products from the cloud provider have been identified. Other products will need to be selected as design progresses.
	CRN-7		The use of Spring which is based on the IoC pattern facilitates unit testing

			using frameworks such as JUnit and Mockito.
		CRN-8	Swagger/OpenAPI chosen to document APIs
CRN-9			No relevant decisions made yet.

8.3.4 Iteration 3: Addressing Reliability and Availability quality attributes

This section presents the results of the activities that are performed in each of the steps of ADD in the third iteration of the design process. While the decisions made in the previous iteration are sufficient to support price changes and querying these price changes from a functional perspective on the backend side, several quality attributes have not yet been addressed. These are the focus of this iteration.

8.3.4.1 Step 2. Establish iteration goal by selecting drivers

Iteration goal: The goal of this iteration is to identify additional structures to support quality attributes, particularly Reliability, Availability and Scalability:

- QA-2: Reliability—Guaranteeing that 100% of the price changes are published and exported successfully
- QA-3: Availability—Uptime SLA
- QA-4: Scalability—Increase query volume without decreasing average latency

In the previous iteration, a new concern was identified, this concern plays an important role with respect to the quality attributes:

- CRN-9: Configuration of cloud provider managed services.

8.3.4.2 Step 3. Choose one or more elements of the system to refine

For this iteration, the elements that are refined are mainly located in the infrastructure part of the solution.

Furthermore, additional refinements are made to the export microservice to support failure scenarios.

8.3.4.3 Step 4. Choose one or more design concepts that satisfy the selected drivers

Design concepts used in this iteration are the following:

Design decisions and location	Rationale and assumptions
Apply tactics for performance: Increase Resources and Maintain Multiple Copies of Computations and Data	Replication is an essential strategy to address reliability and also performance.
Apply tactics for availability: Exception Handling	When one of the microservices fails, the problem needs to be detected and addressed so that the system as a whole continues operating normally.
Apply tactics for availability: Redundant spare	There may be a more generalized failure occurring in the cloud's availability zone where the various microservices are located. This situation also needs to be addressed.

8.3.4.4 Step 5. Instantiate architectural elements, allocate responsibilities and define interfaces

The instantiation design decisions are summarized in the following table:

Design Decision and Location	Rationale
Use Kafka as the data storage mechanism instead of a database for the export microservice.	<p>As described in the previous iteration, price change events that are received by the export microservice are forwarded to the Channel Manager System.</p> <p>In case of communication failures with the Channel Manager System, there is no need to store the events to later retry: Kafka can be used as the storage mechanism (this is described in more detail in step 6).</p> <p>A discarded alternative is to add a database to the export microservice, but this would incur additional costs and would not provide additional benefits.</p>
Configure the container orchestration service to restart microservices in case of failures	<p>Container orchestration services from the cloud provider support restarting containers in case a failure is detected.</p> <p>A discarded alternative is to use an open source solution to orchestrate containers. However, it was previously decided that managed services are preferred for this project.</p>
Establish an infrastructure configuration to support Passive Redundancy	<p>The design of the system allows it to be replicated. This involves creating a duplicate of the infrastructure hosted in a different availability zone or region. When a failure is detected, this replica needs to be started.</p> <p>Configurations of the managed services that support redundancy are therefore selected.</p> <p>A discarded alternative is to manually replicate the whole infrastructure in different availability zones or regions. However, the work involved in setting up and managing such an approach makes it undesirable.</p>
Replicate the Query service	<p>To increase performance, the query microservice is replicated and requests are load balanced.</p>

The results of these instantiation decisions are recorded in the next step.

8.3.4.5 Step 6. Sketch views and record design decisions

Several decisions are made during this iteration to address the drivers listed in step 2. Here we summarize these decisions.

Communication failure with the Channel Management System

In this iteration, a failure scenario of communication between the export microservice and the channel management system is considered (associated with QA-2).

[Figure 8.13](#) shows the interaction that occurs on the export microservice when the event is received. In this scenario, the event is sent to the Channel Management System but the invocation does not succeed (for example, there is a timeout). When this situation occurs, the event that was consumed from the event log is sent back to Kafka. This means that the next time the export microservice polls the event log, it will recover the event that was not sent successfully.

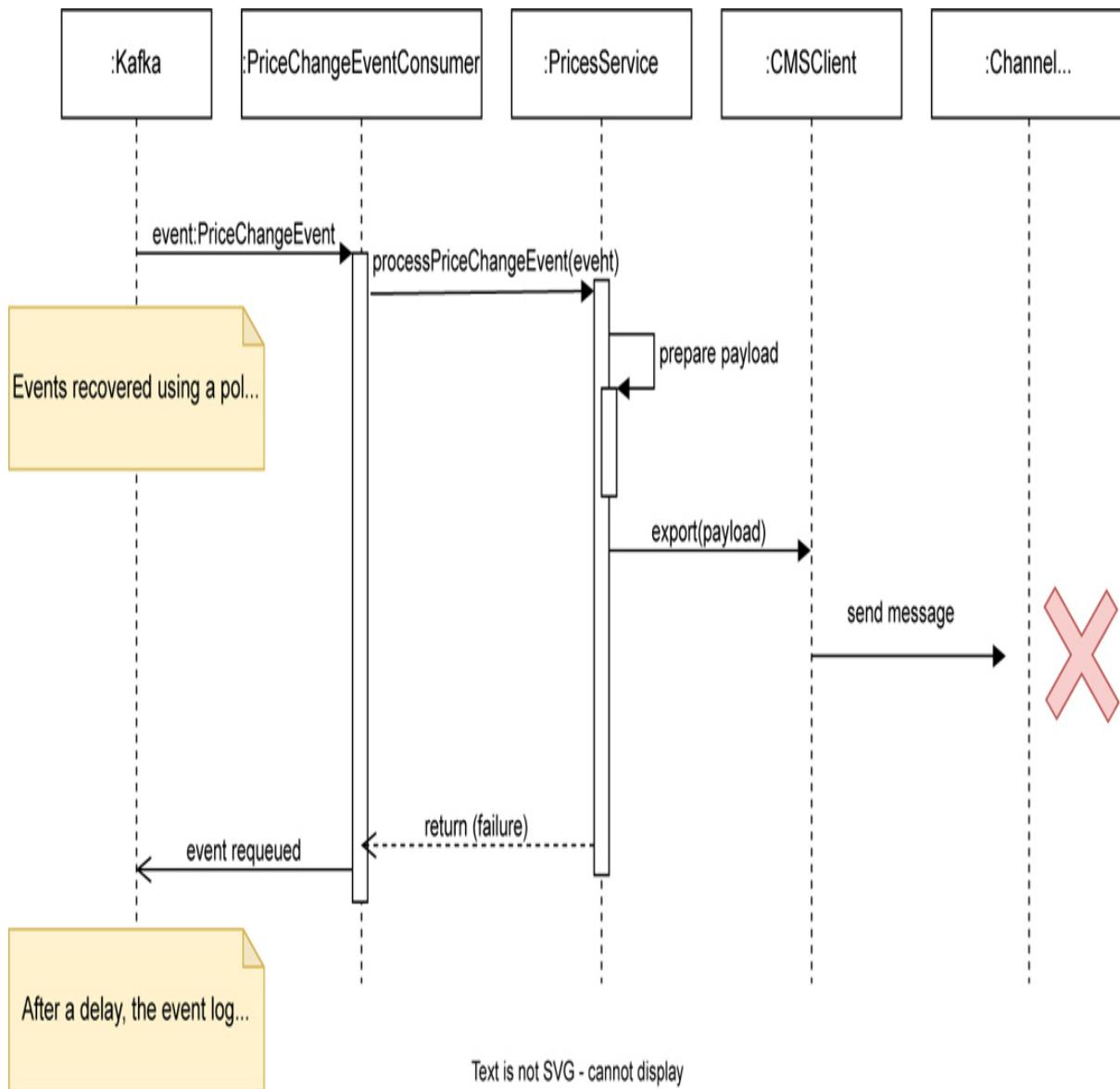
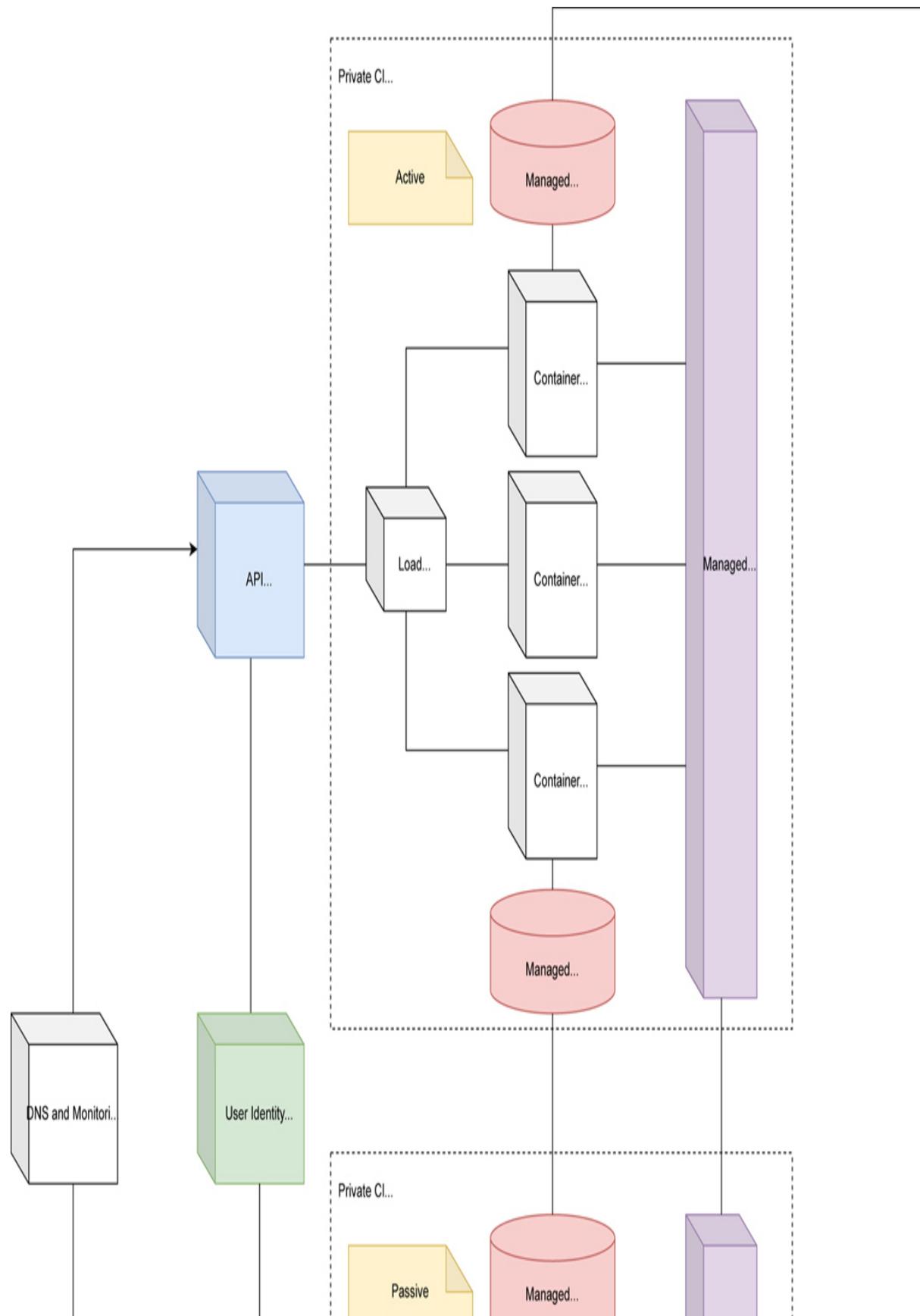


FIGURE 8.13 Sequence diagram of failure scenario on the export side (Key: UML)

Changes in the infrastructure to support quality attributes

Figure 8.14 shows the managed services with the introduction of replication.



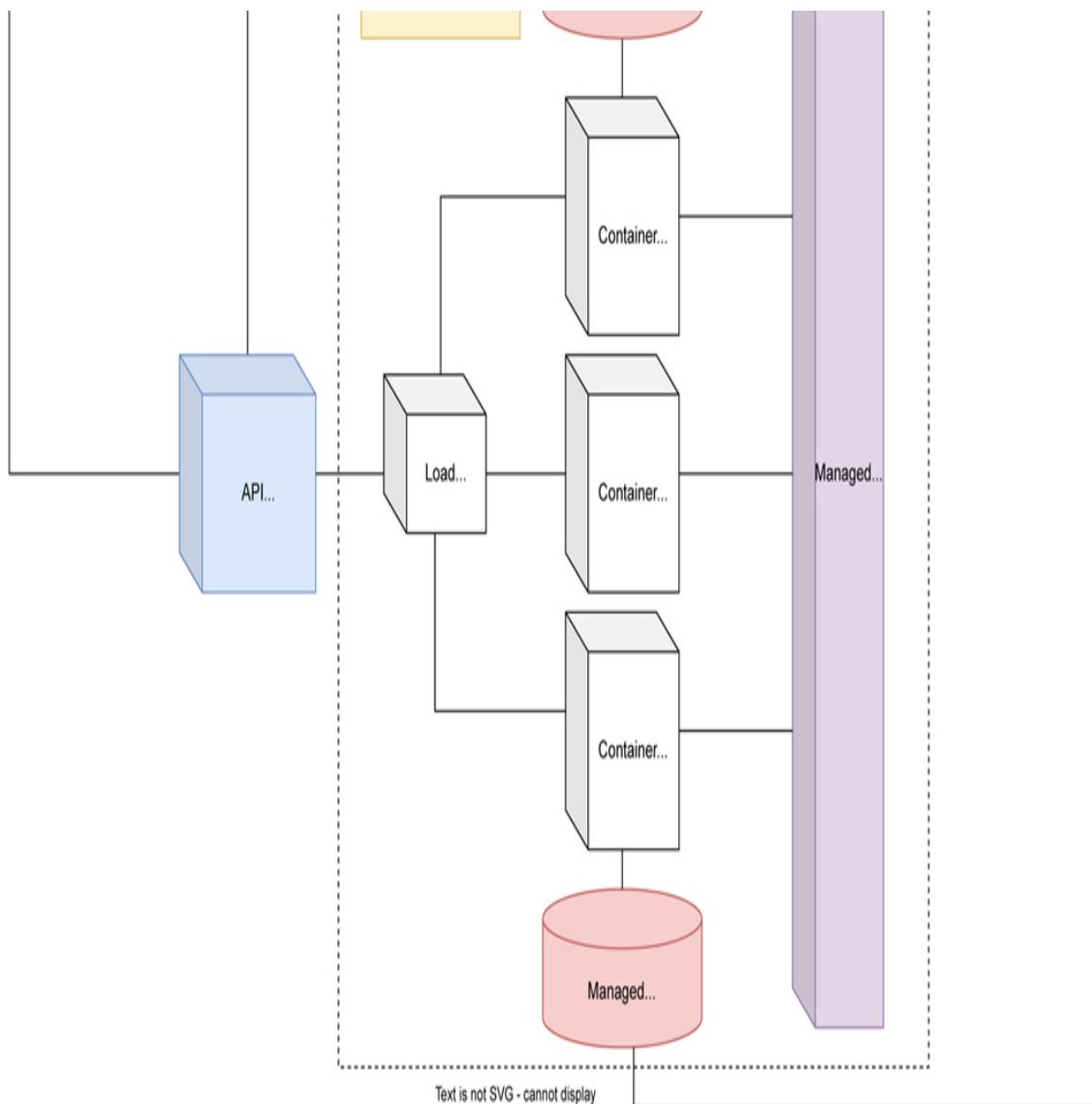


FIGURE 8.14 Refined deployment diagram (Key: UML)

The responsibilities for the elements identified in the previous diagram are the following (only the elements that were updated are described):

Element	Responsibility
DNS and monitoring service	This service monitors the availability of the services in the active replica and, in case a failure is detected, starts up the containers on the passive replica and redirects traffic to the passive replica.
User Identity Management Service	Manage users and provide and validate access tokens used to access APIs. A high availability configuration is selected where the service is replicated across availability zones.
Container Orchestration Service	Provides an execution environment for the containers associated with the different microservices. This service also monitors the health of the containers and, if needed, restarts them in case of failures, contributing to improved availability (QA-3).
Managed Relational Database	This is a database service managed by the cloud provider. A high availability configuration is selected where the database is replicated across availability zones.
Managed Non relational Database	This is a database service managed by the cloud provider. A high availability configuration is selected where the database is replicated across availability zones.
Managed Kafka Service	This is a Kafka service managed by the cloud provider. A high availability configuration is selected where Kafka is replicated across availability zones.

8.3.4.6 Step 7. Perform analysis of current design and review iteration goal and achievement of design purpose

The following table summarizes the status of the different drivers and the decisions that were made during the iteration.

Note that drivers that were completely addressed in the previous iteration are removed from the table.

Not Addressed	Partially Addressed	Completely Addressed	Design decisions made during the iteration
	HPS-1		No additional decisions were made during the iteration
	HPS-2		No additional decisions were made during the iteration
	HPS-4		No additional decisions were made during the iteration
	HPS-5		No additional decisions were made during the iteration
	HPS-6		No additional decisions were made during the iteration
	QA-1		Replication of query microservice improves performance and scalability. Tests to validate the measures associated with the quality attributes remain to be done.
	QA-2		Kafka is highly reliable and the use of a transaction manager to ensure that an event is sent when prices are changed will promote reliability.

	QA-3		Container orchestration addresses failures in containers; replication across availability zones addresses failures in a particular availability zone.
	QA-4		Tests to validate the measures associated with the quality attributes remain to be done.
	QA-5		No additional decisions have been made
	QA-7		No additional decisions have been made
QA-8			No relevant decisions made yet.
CON-3			No relevant decisions made yet.
	CRN-4		A careful analysis considering growth scenarios and cost helps to select technologies that are more future-proof hence avoiding technical debt.
CRN-5			No relevant decisions made yet.
	CRN-6		Specific products from the cloud provider have been identified. Other products will need to be selected as design progresses.
	CRN-9		Initial configurations have been established for the container orchestration service and other managed services (DB and Kafka).

8.3.5 Iteration 4: Addressing development and operations requirements

At some point during the design process, decisions need to be made to satisfy development and operations requirements. This iteration focuses on decisions made to satisfy QA-7 and CRN-5, which are these types of requirements.

8.3.5.1 Step 2. Establish iteration goal by selecting drivers

For this iteration, the architect focuses on satisfying the following requirements:

- *QA-7 (Deployability)*: The application is moved between environments as part of the development process. Moving is performed successfully in at most 4 hours.
- *CRN-5*: Set up continuous deployment infrastructure.

Certain constraints are also considered:

- *CON-3*: Code must be hosted on a proprietary Git-based platform.

8.3.5.2 Step 3. Choose one or more elements of the system to refine

For this scenario, the elements that will be refined are the various microservices. However, elements that haven't been identified previously will be added.

8.3.5.3 Step 4. Choose one or more design concepts that satisfy the selected drivers

Design concepts used in this iteration are the following

Design decisions and location	Rationale and assumptions
Apply the tactic of Managing a Deployment Pipeline	The company wants to automate the tasks needed to put the application in the pre-production and production environments. This can be achieved by setting up a CI/CD Pipeline.
Apply the tactic of managing the deployed system	Once built, a container image needs to be uploaded to a container registry so that it can be pulled and instantiated in the different environments. This is necessary for initial deployment and for updates.
Apply the tactic of Scripting Deployment Commands to automate the instantiation of the infrastructure (infrastructure as code)	It is possible to manually set up all the infrastructure in the cloud provider environment, but this is tedious and error-prone. Furthermore, the infrastructures need to be similar in the different environments. Instead of setting them up manually, they are described in a deployment configuration script that can be used to easily instantiate an environment. Also, changes in the configuration of the infrastructure are easily replicated among the different environments.

8.3.5.4 Step 5. Instantiate architectural elements, allocate responsibilities and define interfaces

The instantiation design decisions are summarized in the following table:

Design Decision and Location	Rationale
Use proprietary Git-based platform support for CI/CD pipelines	<p>Constraint CON-3 states that code must be hosted on a proprietary Git-based platform. Since this platform provides support for CI/CD pipelines, this can be leveraged. Each microservice is managed as an independent project and each one has to include an individual pipeline.</p> <p>Discarded alternatives are: setting up a separate CI Server using tools such as Jenkins.</p>
Use Cloud provider's Container Registry to host docker images.	<p>Since CON-6 states that a cloud-native approach should be favored, it makes sense to leverage the support that the selected cloud provider offers for hosting container images.</p> <p>Discarded alternatives include Docker Hub because it would require the payment of an additional subscription which is more expensive than the costs of the cloud provider's container registry.</p>
Use cloud provider's support for infrastructure as code to automate instantiation of environments	<p>Since CON-6 states that a cloud-native approach should be favored, it makes sense to leverage the support that the selected cloud provider offers for describing infrastructure as code.</p> <p>Discarded alternatives include Terraform as it requires managing external tools and, furthermore, the costs incurred by the use of the cloud provider's infrastructure as code features is not significant.</p>

The results of these instantiation decisions are recorded in the next step.

8.3.5.5 Step 6. Sketch views and record design decisions

[Figure 8.15](#) shows how each microservice is structured as an individual project which is hosted in an individual repository:

Microservice Project Repository

Source and test code + resources

Maven POM

Dockerfile

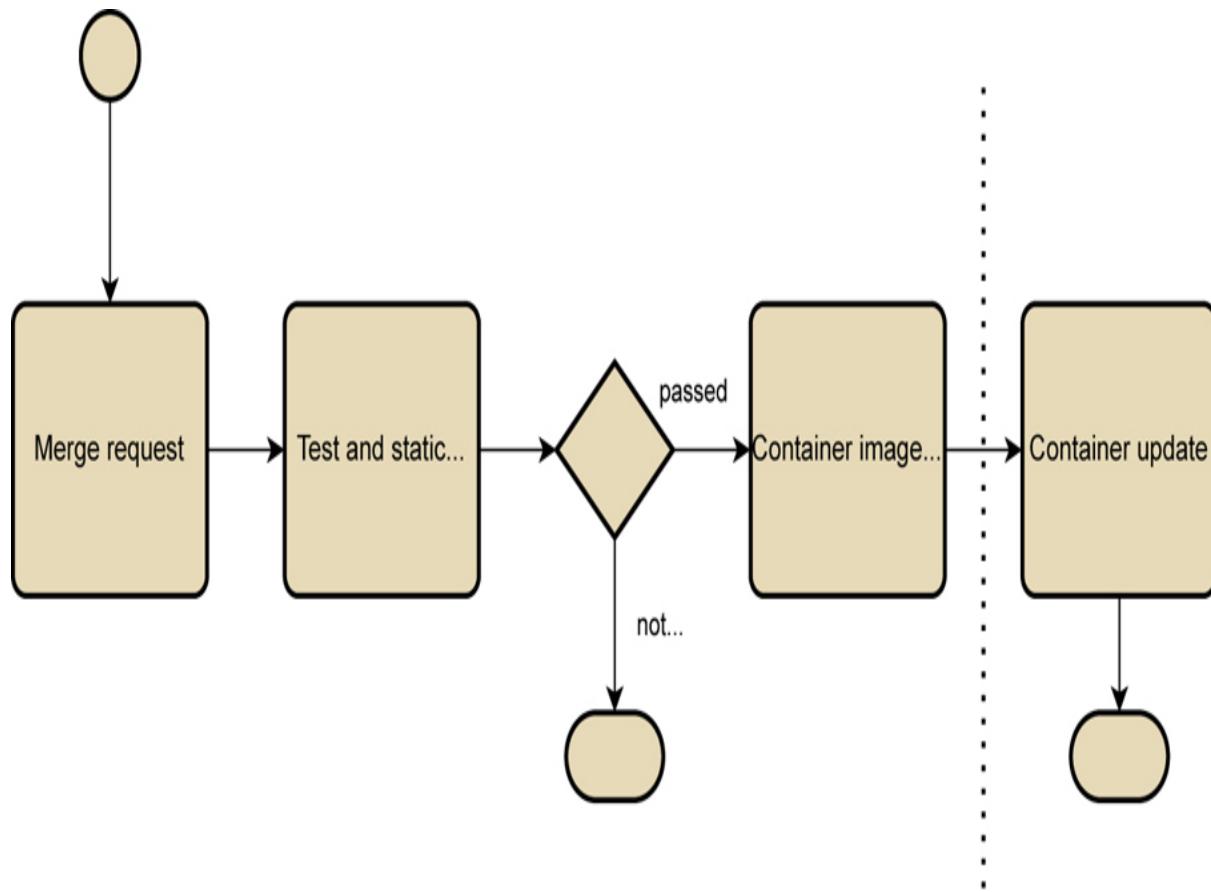
Pipeline + cloud descriptors

Text is not SVG - cannot display

FIGURE 8.15 Package diagram showing structure of repositories (Key: UML)

Element	Responsibility
Microservice project repository	A project for one microservice which is hosted in a single repository
Source and test code + resources	Source and test code along with resources.
Maven POM	Maven descriptor used to automate the build process.
Dockerfile	Used to build Docker container images.
Pipeline + cloud descriptors	Descriptors which provide information for the CI/CD pipeline steps and for configuring the cloud resources where the microservice is executed.

[Figure 8.16](#) shows the sequence of tasks that triggers the update a microservice:



Git-based pla...

Text is not SVG - cannot display

Cloud...

FIGURE 8.16 Activity diagram for the integration and deployment process (Key: UML)

A merge request of a branch with new changes to the main branch triggers the execution of tests and static code analysis (the pipeline descriptor triggers Maven tasks). If the previous step passes successfully, a container image is generated using the Dockerfile and uploaded to the container registry (this is where the transition from the Git-based platform to the cloud environment occurs). Once uploaded, the container is updated by the orchestrator in the cloud provider's environment.

8.3.5.6 Step 7. Perform analysis of current design and review iteration goal and achievement of design purpose

In this iteration, important design decisions have been made to address QA-7 and CRN-5. The following table summarizes the status of the different drivers and the decisions that were made during the iteration.

Note that drivers that were completely addressed in the previous iteration are removed from the table.

Not Addressed	Partially Addressed	Completely Addressed	Design decisions made during the iteration
	HPS-1		No additional decisions were made.
	HPS-2		No additional decisions were made.
	HPS-4		No additional decisions were made.
	HPS-5		No additional decisions were made.
	HPS-6		No additional decisions were made.
	QA-1		No additional decisions were made.
	QA-2		No additional decisions were made.
	QA-3		No additional decisions were made.
	QA-4		No additional decisions were made.
	QA-5		No additional decisions were made.
		QA-7	The use of docker and externalization of configuration values plus the establishment of a CI/CD pipeline allows this requirement to be satisfied.
QA-8			No relevant decisions made yet.

	CON-3	The code is now hosted on a Git-based platform.
	CRN-4	No relevant decisions are made during this iteration.
	CRN-5	An infrastructure to support continuous integration and deployment has been established. Further decisions need to be made to manage scenarios such as rollbacks.
	CRN-9	Additional configurations to cloud provider managed services were made, particularly to the container registry.

At this point it was determined that sufficient decisions had been made to proceed directly to implementation, without needing further ADD iterations for this initial release of the system. As described in CON-4, the system was to have a working MVP within two months, and so it was important to transition quickly to an initial implementation.

8.5 Summary

In this chapter we presented an example of using ADD to design a greenfield system in a mature domain. We first illustrated 3 iterations with different foci: addressing the general concern of structuring the application, addressing functionality, and addressing quality attribute scenarios. We then illustrated a 4th iteration focused on supporting development and operations requirements.

The example follows the roadmap discussed in [section 4.3.1](#), and it is interesting to note that the application is structured using the CQRS pattern implemented as a set of microservices but that the microservices themselves are structured internally using a reference architecture. Also, the selection of externally developed components, in this

case frameworks and products, both open source and from a cloud platform, was performed across the different iterations. The example also illustrates how the design results in a system that is functional early in the development process, which aligns with agile development practices. Finally, the example also illustrates that new architectural concerns appear as the design progresses. It should be noted that CRN-4, the concern associated with avoiding introducing technical debt, is never completely addressed.

This example was structured to show how concerns, primary user stories, and quality attribute scenarios can be addressed as part of architectural design. Also, we illustrate that architectural design sometimes requires a certain amount of detail and it is not only a “high level” design. The design is not completed with the iterations that were described and additional iterations would be necessary to satisfy the drivers that were not completely addressed. However, the decisions made within the four iterations described are sufficient to demonstrate how ADD is conducted to generate the initial system MVP. Finally, it must be noted that prototypes or analyses need to be made to ensure that the measures associated with the QA scenarios are satisfied.

8.6 Further Reading

An online design concepts catalog provides descriptions and bibliographical references of the different design concepts used in this case study.

8.7 Discussion questions

1. A confirmation mechanism for price publication needs to be added to the system so that the user interface can show some icons that signal whether a price change is ready to be queried or has been exported to the Channel Management System. What changes would you make to the architecture to accommodate that requirement?
2. How would you modify the design to support QA-8 (Monitorability)?
3. Several quality attributes were addressed by using cloud managed solutions. Can all quality attributes be satisfied like this?
4. Suppose that new hotels are added to the system and this needs to be communicated to other systems in the company. What changes would you make to the architecture to accommodate this requirement?
5. Considering the design decisions that were made during the four iterations, can you think of a different ordering for them? For example, can deployment and operations design decisions be made prior to the ones made in the earlier iterations?
6. Consider CON-5: The system must interact initially with existing systems through REST APIs but may need to later support other protocols. What changes would you make to the design of the system, if any, to accommodate this constraint?

9. Case Study - Digital Twin Platform

With Serge Haziyev, Yaroslav Pidstryhach and Rodion Myronov

We now present an extended design example of using ADD in a greenfield system for an emerging domain, that of Digital Twins in an Industry 4.0 context. At the time of writing, this domain was still relatively new and rapidly evolving. This domain requires a combination of multiple disciplines such as IoT, cloud computing, big data and analytics, AI/ML, Extended Reality (XR), simulation, advanced automation, and often robotics. Such a vast number of domains is beyond the expertise of a single architect, and this case study exemplifies how a team of architects from different disciplines can participate in the design of a system (as discussed in [Chapter 12](#)). Here, none of the architects could solely rely on their experience alone to guide them. Instead, they had to cooperate and leverage design concepts and best practices from each discipline, as we will now describe.

Why read this chapter?

In the last chapter we said that people learn best from examples. But having more than one example is even better! Here we give you another example of design using ADD in a very different, and challenging, context. The scale of the system here is much larger than the one presented in [Chapter 8](#), and this introduces the need to consciously align specific considerations and design choices made in each iteration with the overall system drivers. In this chapter we hope to convince you that, despite this added complexity, ADD works for architectural design at *all* scales.

9.1 Business Case

This client for this case study is a large food producer with dozens of production plants, many of them containing several production lines. The company is committed to developing innovative solutions with a mission to ensure the advancement of food security and promotion of sustainability. For them, creating a smart factory and automating industrial processes is crucial in addressing the shortage of skilled workers and maintaining affordable prices for consumers.

One of the key investments of the company is creation of a digital twin platform (a computational platform with digital representations of assets and processes) that will enable a range of use cases, from simple ones like remote monitoring of industrial processes across plants to more complex ones, such as process simulation and autonomous decision-making to adapt to a changing business environment as quickly as possible. The platform should provide the capabilities to implement various use cases that contribute to the goal of creating a smart factory.

The marketecture diagram (an informal depiction of the system's structure) shown in [Figure 9.1](#) represents the

desired solution from a functional perspective.

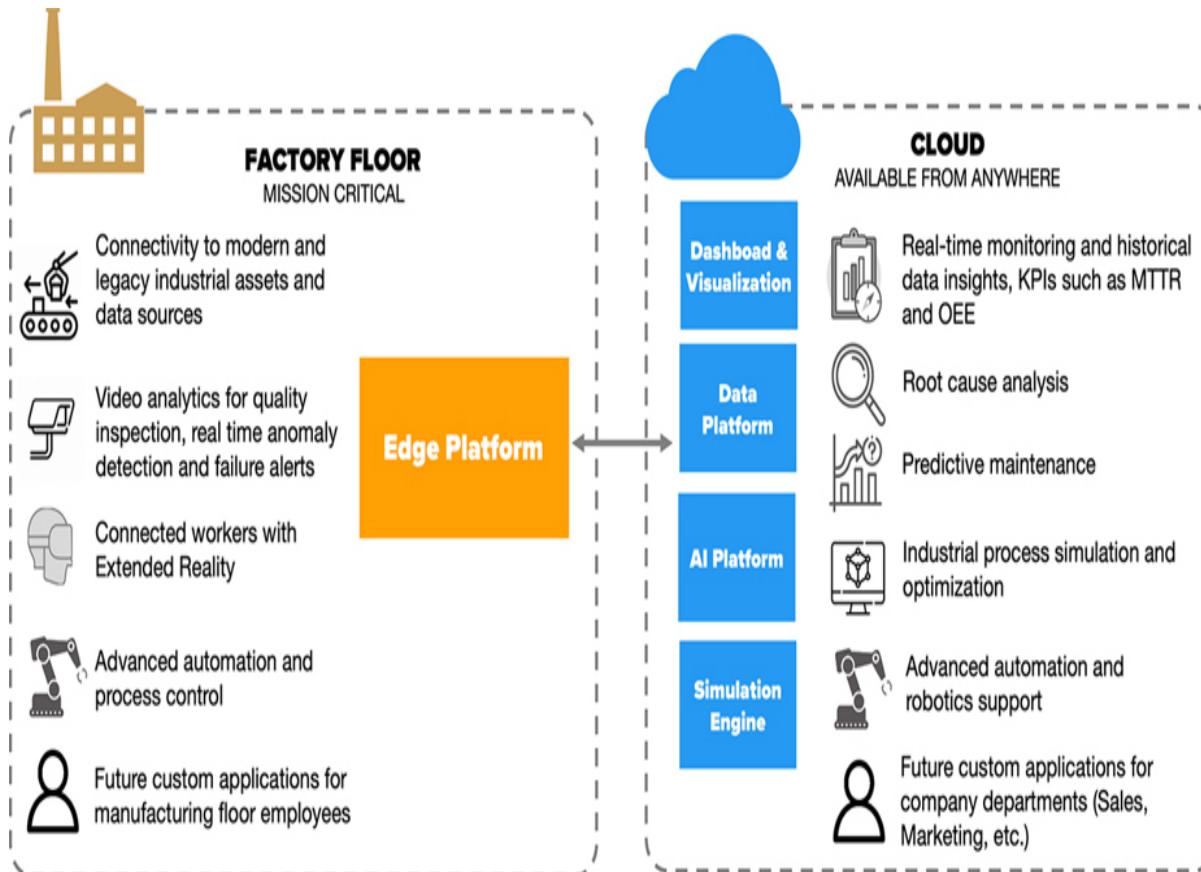


FIGURE 9.1 Marketecture Diagram for the *Digital Twin Platform*

9.2 System Requirements

For requirements elicitation activities, the Digital Twin Maturity Model (DTMM), shown in [Figure 9.2](#) was used to identify use cases corresponding to each of the five digital twin maturity stages:

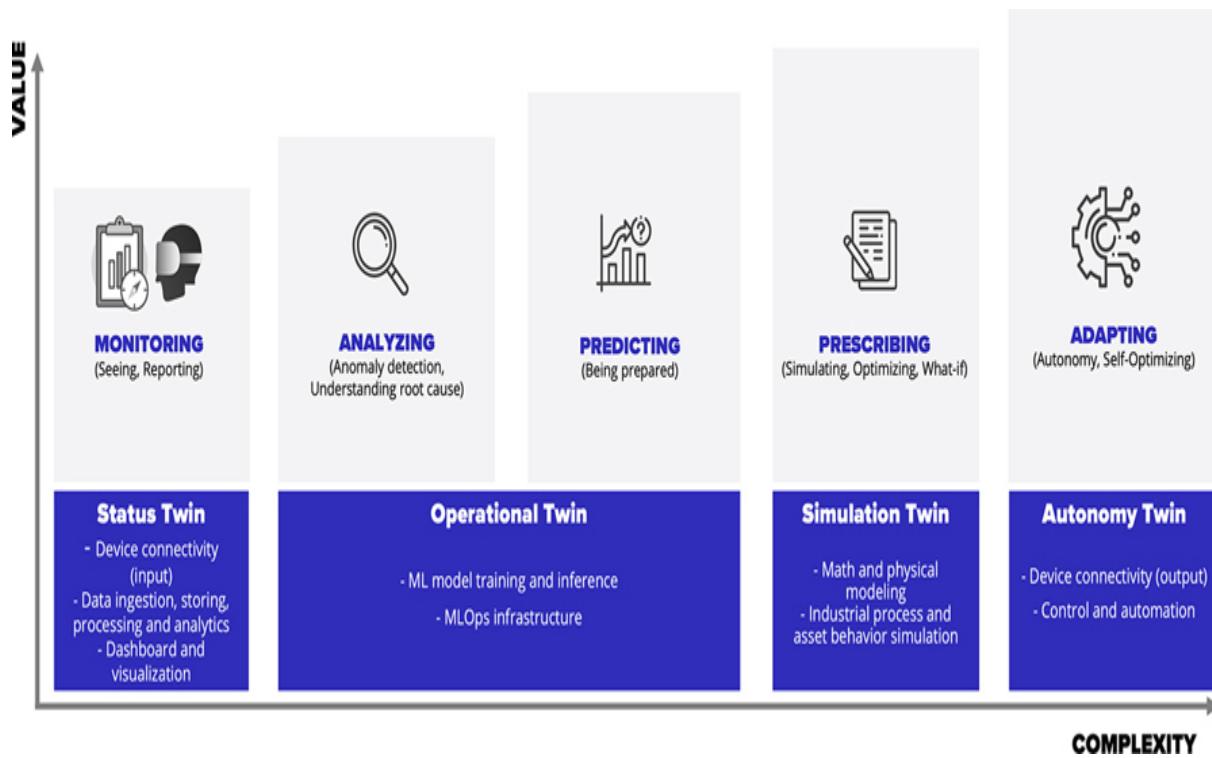


FIGURE 9.2 *Digital Twin Maturity Model*

Each stage represents a group of capabilities that the platform should support to provide value to its users. During the requirement elicitation process, base use cases were chosen for each stage, as this technique allows for a holistic view of the system's capabilities rather than focusing on just a part of it. Some of the use cases chosen are abstract and require further development and elicitation into more concrete forms, as innovative solutions typically require exploration and experimentation. Therefore, the platform's goal is to enable both the identified use cases and similar future ones.

The following is a summary of the most important requirements collected. These are a set of primary use cases, a set of quality attribute scenarios, and a set of constraints.

9.2.1 Use case model

The following table presents the primary use cases for the system and their relationship with the DTMM.

Use Case	Description	DTMM Stage
UC-1: Real-time monitoring and historical data insights	As a Global Operator, I can see information such as OEE (Overall Equipment Effectiveness) about any industrial process on any plant. The data should be collected from equipment in real-time and stored in the cloud.	Monitoring
UC-2: Quality inspection automation	As a Quality Control Operator, I can control the quality of production via sensors (including optical). The system should immediately generate alerts in case of detection of anomalies.	Analyzing
UC-3: Predictive maintenance and XR for maintenance activities	As a Maintenance Engineer, I can get predictions on equipment failures and diagnose them by accessing historical and real-time information. The information should be accessible through a hands-free extended reality device.	Predicting
UC-4: Process simulation	As a Process Engineer, I can find optimal parameters for equipment settings based on raw materials and desired quality of output. The parameters should be derived from simulation.	Prescribing

UC-5: Advanced automation	As an Automation Engineer, I want to automate operational decision-making, so that the plant can adapt to changing conditions (such as orders, raw materials, or power supply) without human intervention.	Adapting
---------------------------	--	----------

9.2.2 Quality attribute scenarios

The following is a list of quality attribute scenarios for the system.

ID	Quality Attribute	Scenario	Associated Use Case
QA-1	Availability	Critical system functions for the production floor must remain 100% available in the event of a cloud or network outage. Should such an event occur, data should be cached at the edge for at least 8 hours.	UC-2
QA-2	Security	An unauthorized access attempt is made. The attempt is denied, logged, and the system administrator is alerted within 15 minutes of the attempted breach.	All Use Cases
QA-3	Deployability	A system deployment is initiated. The deployment is fully automated and supports at least development, test, and production environments.	All Use Cases
QA-4	Performance	100,000 data points (where data point is 120 bytes in average) arrive at the system per second. The system processes them all within one second.	All Use Cases
QA-5	Performance	The real-time dashboard for Global Operators is refreshed automatically, at 15 second intervals, with < 15 sec data latency.	UC-1

QA-6	Performance	Industrial processes, coupled with real-time anomaly detection, are continuously visually inspected. Inspection results are received within 5 seconds.	UC-2
QA-7	Performance	When a data analyst requests an ad-hoc SQL-like queries for raw and aggregated historical data (multi-relational data that can include various equipment from different plants), 95% of queries return results within 30 seconds.	UC-3, 4, 5
QA-8	Extensibility	New devices and sensors can be added to the system at runtime, with no interruption of ongoing data collection and system functionality.	All Use Cases
QA-9	Extensibility	New applications (e.g. Production, Quality, Supply Chain, Sales) can be created building on existing services without the need to re-architect or change the system's main components (such as Edge, Data and AI Platforms).	All Use Cases
QA-10	Observability	Key performance metrics and error logs are collected, aggregated, and visualized in a monitoring dashboard with a maximum delay of 5 seconds.	All Use Cases

9.2.3 Constraints

The following is a list of constraints that were levied on the system.

ID	Constraint
CON-1	The system shall use existing company cloud infrastructure on AWS
CON-2	Use cloud-native strategy (prefer microservices, containers and managed services) for building scalable applications
CON-3	When selecting cloud services give preference to the services that have the lowest and most predictable costs.
CON-4	NIST's IoT Cybersecurity Capabilities must be implemented to manage cybersecurity risks

9.3 The Design Process

Now that we have enumerated the architecturally significant requirements, we are ready to begin the first iteration of ADD.

9.3.1 ADD step 1 - Review Inputs

The first step of the method involves reviewing the inputs. They are summarized in the following table:

Category	Details																														
Design purpose	<p>This is a greenfield system in a novel domain. The organization will perform development following an agile process with short iterations to quickly receive feedback and continue modifying the system. At the same time, an architectural design is needed to make conscious decisions to satisfy architectural drivers and avoid unnecessary rework.</p>																														
Primary functional requirements	<p>The use cases outlined in Section 9.2.1 have been reviewed. Of these, UC-1, UC-2, and UC-3 have been selected as primary. The other use cases, specifically UC-4 and UC-5, are slated for inclusion in future versions of the platform.</p>																														
Quality attribute scenarios	<p>The following table illustrates the priority of the quality attribute scenarios, as ranked by the customer and architect (as discussed in Chapter 2, section 2.4.2). Of course, quality attribute scenarios with lower priorities exist but they are not shown here.</p> <table border="1" data-bbox="486 1115 1383 1915"> <thead> <tr> <th data-bbox="486 1115 736 1326">Scenario ID</th><th data-bbox="736 1115 1046 1326">Importance to customer</th><th data-bbox="1046 1115 1383 1326">Difficulty of implementation according to architect</th></tr> </thead> <tbody> <tr> <td data-bbox="486 1326 736 1396">QA-1</td><td data-bbox="736 1326 1046 1396">High</td><td data-bbox="1046 1326 1383 1396">Medium</td></tr> <tr> <td data-bbox="486 1396 736 1465">QA-2</td><td data-bbox="736 1396 1046 1465">High</td><td data-bbox="1046 1396 1383 1465">High</td></tr> <tr> <td data-bbox="486 1465 736 1535">QA-3</td><td data-bbox="736 1465 1046 1535">Medium</td><td data-bbox="1046 1465 1383 1535">Medium</td></tr> <tr> <td data-bbox="486 1535 736 1605">QA-4</td><td data-bbox="736 1535 1046 1605">High</td><td data-bbox="1046 1535 1383 1605">Medium</td></tr> <tr> <td data-bbox="486 1605 736 1674">QA-5</td><td data-bbox="736 1605 1046 1674">Medium</td><td data-bbox="1046 1605 1383 1674">Medium</td></tr> <tr> <td data-bbox="486 1674 736 1744">QA-6</td><td data-bbox="736 1674 1046 1744">High</td><td data-bbox="1046 1674 1383 1744">High</td></tr> <tr> <td data-bbox="486 1744 736 1814">QA-7</td><td data-bbox="736 1744 1046 1814">Medium</td><td data-bbox="1046 1744 1383 1814">Medium</td></tr> <tr> <td data-bbox="486 1814 736 1883">QA-8</td><td data-bbox="736 1814 1046 1883">High</td><td data-bbox="1046 1814 1383 1883">Medium</td></tr> <tr> <td data-bbox="486 1883 736 1915">QA-9</td><td data-bbox="736 1883 1046 1915">High</td><td data-bbox="1046 1883 1383 1915">Medium</td></tr> </tbody> </table>	Scenario ID	Importance to customer	Difficulty of implementation according to architect	QA-1	High	Medium	QA-2	High	High	QA-3	Medium	Medium	QA-4	High	Medium	QA-5	Medium	Medium	QA-6	High	High	QA-7	Medium	Medium	QA-8	High	Medium	QA-9	High	Medium
Scenario ID	Importance to customer	Difficulty of implementation according to architect																													
QA-1	High	Medium																													
QA-2	High	High																													
QA-3	Medium	Medium																													
QA-4	High	Medium																													
QA-5	Medium	Medium																													
QA-6	High	High																													
QA-7	Medium	Medium																													
QA-8	High	Medium																													
QA-9	High	Medium																													

	QA-10	Medium	High
Constraints	See section 9.2.3		

9.3.2 Iteration 1: Reference Architecture and Overall System Structure

This section presents the results of the activities that are performed in each of the steps of the ADD method in the first iteration of the design process.

9.3.2.1 Step 2 - Establish iteration goal by selecting drivers

Since this is the first iteration in the design of a greenfield system, the iteration goal is to establish an initial overall structure for the system. The architect must keep in mind all of the drivers, but primarily the constraints and high priority quality attributes. Note that in this chapter we follow a variant of the roadmap we presented in [chapter 4](#), modified to accommodate the very large scale of this system.

9.3.2.2 Step 3 - Choose one or more elements of the system to refine

As this is the initial iteration, we must refine the entire system. This system, as it has been envisioned, is expressed in the marketecture diagram ([Figure 9.1](#)) which suggests two major components to refine: Factory Floor (Edge) and Cloud.

9.3.2.3 Step 4 - Choose one or more design concepts that satisfy the selected drivers

In this iteration, design concepts are selected based on the major components of the system: Edge and Cloud.

Design decisions and location	Rationale
Select a public cloud provider.	Based on CON-1 the system should leverage existing cloud infrastructure on AWS
Choose the PaaS and FaaS service models.	<p>In the case of Platform as a Service (PaaS) and Function as a Service (FaaS) resource consumption models, the cloud provider manages the infrastructure, facilitating the adoption of a cloud-native strategy (CON-2).</p> <p>The system can still utilize Infrastructure as a Service (IaaS, i.e. virtual machines) for components that are not supported by PaaS or FaaS services.</p> <p>Please refer to “7.1.2: Models of resource provision” for more details.</p>
Use Edge Computing.	<p>The following considerations are taken into account:</p> <p>Network reliability: When dealing with mission-critical applications network reliability becomes a crucial factor to allow local processing even when there is intermittent or no connectivity to the cloud. At least two use cases UC-2 (quality inspection automation) and UC-5 (advanced automation) imply mission-criticality.</p> <p>Latency reduction: Edge computing processes data closer to the source, resulting in lower latency. This is particularly important in applications that demand real-time data processing and decision-making, such as computer vision (UC-2).</p>



	<p>Bandwidth optimization: By processing data at the edge, only relevant information is sent to the cloud, reducing the amount of bandwidth required. This can help optimize network usage and decrease operational costs (UC-1, UC-2).</p> <p>The alternative, connecting the devices directly to the cloud, compromises reliability and is typically not used in manufacturing.</p>
<p>Use the Digital Twin Maturity Model (DTMM) as a reference model to identify capabilities that can be matched with architectural elements.</p>	<p>The DTMM (see Figure 9.1) can serve as a reference model that guides the mapping of desired platform capabilities to architectural elements. Here is a breakdown of capabilities associated with the types of digital twins and stages of the DTMM.</p> <p>Status Twin—Monitoring stage capabilities (UC-1):</p> <ul style="list-style-type: none"> ▪ Input and output device connectivity ▪ Data ingestion, storing, processing and analytics ▪ Dashboard and visualization <p>Operational Twin—Analyzing (UC-2) and Predicting (UC-3) stages capabilities:</p> <ul style="list-style-type: none"> ▪ ML model training and inference ▪ MLOps infrastructure <p>Simulation Twin—Prescribing (UC-4) stage capabilities:</p> <ul style="list-style-type: none"> ▪ Math and physical modeling ▪ Industrial process and asset behavior simulation <p>Autonomous Twin—Autonomy (UC-5) stage capabilities:</p>

- | | |
|--|---|
| | <ul style="list-style-type: none">▪ Output device connectivity▪ Control and automation <p>Cross-cutting capabilities:</p> <ul style="list-style-type: none">▪ Access control (QA-2)▪ Future custom applications (shown in the marketecture diagram) |
|--|---|

9.3.2.4 Step 5 - Instantiate architectural elements, allocate responsibilities and define interfaces

The instantiation design decisions considered and made are summarized in the following table:

Design Decision and Location	Rationale														
<p>Instantiate Edge elements based on selected capabilities from Step 4</p>	<p>From the list of capabilities in the DTMM identified in step 4, a subset of capabilities deemed to be most important is selected and instantiated as Edge Elements (components and subsystems):</p> <table border="1" data-bbox="528 551 1372 1769"> <thead> <tr> <th data-bbox="528 551 943 777">Selected Capabilities</th><th data-bbox="943 551 1372 777">Edge element</th></tr> </thead> <tbody> <tr> <td data-bbox="528 777 943 920">Future custom applications</td><td data-bbox="943 777 1372 920">Apps</td></tr> <tr> <td data-bbox="528 920 943 1064">Access control</td><td data-bbox="943 920 1372 1064">Access Control</td></tr> <tr> <td data-bbox="528 1064 943 1208">Control and automation</td><td data-bbox="943 1064 1372 1208">Control & Automation</td></tr> <tr> <td data-bbox="528 1208 943 1431">Data ingestion, storing, processing and analytics</td><td data-bbox="943 1208 1372 1431">Data Store & Analytics</td></tr> <tr> <td data-bbox="528 1431 943 1575">ML inference</td><td data-bbox="943 1431 1372 1575">ML Models & Inference</td></tr> <tr> <td data-bbox="528 1575 943 1769">Input and output device connectivity</td><td data-bbox="943 1575 1372 1769">Input / Output</td></tr> </tbody> </table>	Selected Capabilities	Edge element	Future custom applications	Apps	Access control	Access Control	Control and automation	Control & Automation	Data ingestion, storing, processing and analytics	Data Store & Analytics	ML inference	ML Models & Inference	Input and output device connectivity	Input / Output
Selected Capabilities	Edge element														
Future custom applications	Apps														
Access control	Access Control														
Control and automation	Control & Automation														
Data ingestion, storing, processing and analytics	Data Store & Analytics														
ML inference	ML Models & Inference														
Input and output device connectivity	Input / Output														
<p>Instantiate Cloud elements based on</p>	<p>A subset of the capabilities identified in step 4 are instantiated as Cloud Elements (components and subsystems):</p>														

selected capabilities
from Step 4

Selected capabilities	Cloud element
Access control	Centralized Access Control

	Device management	Device Management
	Connectivity with edge	IoT Message Broker
	Data ingestion, storing, processing and analytics	Data Platform
	Dashboard and visualization	Dashboard
	ML model training and inference on Cloud, MLOps infrastructure	AI Platform
	ML model training and inference on Cloud, MLOps infrastructure	AI Models
	Industrial process and asset behavior simulation	Simulator
	Math and physical modeling	Math & Physical Models
	Future custom applications	Apps

Due to the complexity of this system, defining precise functionality and interfaces is not possible in this early iteration; it is left for later iterations.

9.3.2.5 Step 6 - Sketch views and record design decisions

The diagram in [Figure 9.3](#) shows the resulting reference architecture based on these instantiation decisions:

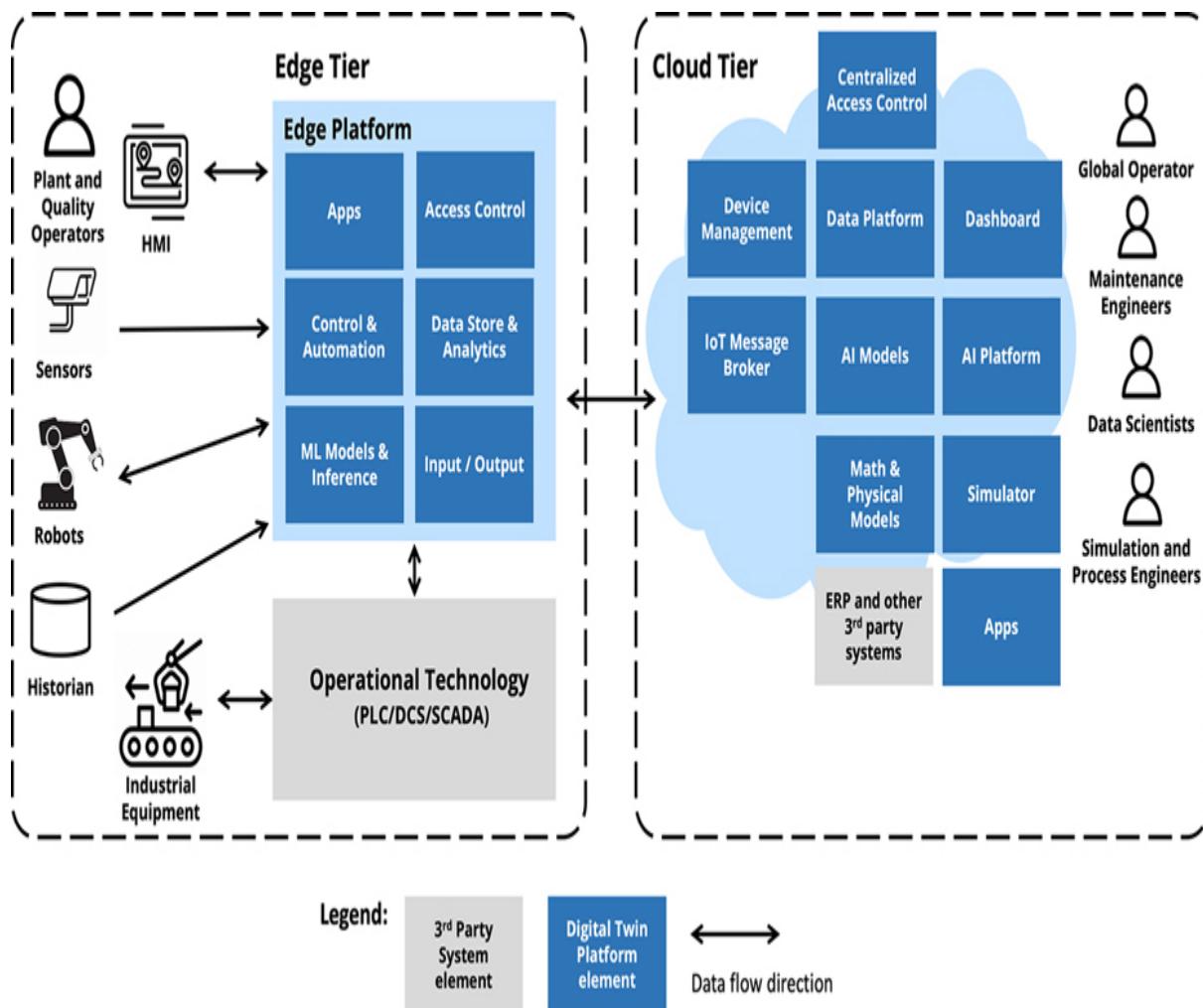


FIGURE 9.3 Digital Twin Platform reference architecture

A summary of each element's responsibilities in the Edge Platform presented in [Figure 9.3](#) is recorded as follows:

Edge element	Responsibility
Apps	Applications for the employees of the manufacturing floor. This is available to users irrespective of connectivity with the cloud
Access Control	Manage and enforce the policies that determine which users and devices are authorized to access and interact with the system at the plant level
Control & Automation	Execute industrial control algorithms such as MPC (Model Predictive Control) or PID (Proportional Integral Derivative); orchestrate and automate industrial processes with decision-making on the edge device
Data Store & Analytics	Local data storage on the edge device, allowing for the storage of both raw and processed data. Real-time data processing, such as filtering, aggregation, and transformation.
ML Models & Inference	Deployment and execution of machine learning (ML) models directly on edge device
Input / Output	Gather data from different sources such as industrial equipment, PLC (Programmable Logic Controller), SCADA (Supervisory Control and Data Acquisition) systems, sensors, and other devices using industrial protocols like OPC UA (Open Platform Communications United Architecture - a data exchange standard for industrial communication).

Send commands to actuators or other control elements within the industrial environment to enable automation of industrial processes.

And the next table describes the elements and their responsibilities for the Cloud part:

Element	Responsibility
Centralized Access Control	Manage and enforce the policies that determine which users and devices are authorized to access and interact with the system globally at the organization level
Device Management	A single point of control for all edge deployments: device provisioning, configuring, monitoring and updating edge software
IoT Message Broker	A high throughput pub/sub messaging component that securely transmits messages to and from Edge instances with low latency
Data Platform	Ingest, store, process and analyze data from various sources, including sensors, edge devices and 3 rd party software such as ERP (Enterprise Resource Planning), MES (Manufacturing Execution System), etc.
Dashboard	A user interface that visualizes and consolidates real-time and historical data, provides means for data analytics and insights
AI Platform	A set of tools to build, train, test and deploy AI models (using mostly machine learning methods)
AI Models	Trained AI models for various tasks (computer vision,

	<p>predictive maintenance, optimization, automation, etc.), for Cloud or Edge inference</p>
Simulator	<p>Realistic virtual environments and “what-if” scenarios that accurately represent real-world conditions, enabling testing and optimization of processes and strategies without impacting actual physical assets</p>
Math & Physical Models	<p>Representation of the underlying physics, dynamics, and behavior of assets, processes, or systems utilizing physical laws, and domain-specific knowledge</p>
Apps	<p>Applications for company departments (and potentially for business partners) that leverage the platform capabilities</p>

9.3.2.6 Step 7. Perform analysis of current design and review iteration goal and achievement of design purpose

The decisions made in this iteration address important early considerations affecting the overall system structure. Additional design decisions will need to be made in later iterations to further decompose the elements, select candidate technologies and provide more details on how use cases and quality attributes will be supported.

The following table summarizes the design progress using the Kanban board technique discussed in 3.8.2.

Not Addressed	Partially Addressed	Completely Addressed	Design decisions made during the iteration
	UC-1		Introduced Edge Platform, Data Platform and Dashboard elements. No detailed decisions of technologies to use have been made.
	UC-2		Use Edge computing with ML inference. No detailed decisions on the technology have been made.
	UC-3		Introduced Data Platform, AI Platform, AI Models elements. No detailed decisions of technologies to use and XR device integration have been made.
	UC-4		Introduced Simulator, Math & Physical Models elements. No detailed decisions on the technology have been made.
	UC-5		Introduced Edge (Control & Automation), AI Platform, AI Models elements. No detailed decisions of technologies have been made.
	QA-1		Use Edge computing for critical functions. No detailed decisions on the technology have been made.
	QA-2		Introduced Access Control on the Edge and Centralized Access Control on the Cloud. No detailed decisions on the technology have been made.
QA-3 QA-4 QA-5			To be addressed in the following iterations
	QA-6		Use Edge computing with ML inference. No detailed decisions on the technology have been made.
	QA-7		Introduced the Data Platform element. No detailed decisions of technologies to use have been made.
QA-8			To be addressed in the following iterations

	QA-9		Introduced Apps elements on the Edge and Cloud. No detailed decisions on the technology have been made.
QA-10			To be addressed in the following iterations
	CON-1		Confirmed that the type of Cloud is Public
	CON-2		The primary Cloud resource consumption models and PaaS and FaaS
CON-3 CON-4			No relevant decisions made

9.3.3 Iteration 2: Refinement of IIoT Elements

The second iteration of this design is focused on the IIoT (Industrial IoT) aspects of the system and the selection of technologies to support system requirements.

9.3.3.1 Step 2. Establish iteration goal by selecting drivers

The following tables identify the chosen drivers, based on what was identified in [section 9.2](#), to support the goal of this iteration. Additionally, for each driver, specific considerations with respect to the focus of the iteration are discussed.

Use cases

Architectural driver	Specific IIoT considerations
UC-1: Real-time monitoring and historical data insights	Although this use case pertains to the Global operator, the data originates from, and is forwarded by, an Edge computing system.
UC-2: Quality inspection automation	At least some components of the quality inspection automation system must be deployed on the Edge, such as the sensing device.

Quality attributes

Architectural driver	Specific IIoT considerations
QA-1: Availability	The availability of the system in an offline mode, or in the event of limited Internet connectivity, is a key aspect of Edge solutions, and a critical architectural driver for the Edge segment. It produces a host of indirect requirements, such as sufficient local storage, data synchronization algorithms, and the ability to make local decisions on the Edge that are critical for the operation of the industrial system.
QA-2: Security	In IoT systems, security through hardware is critical, and is typically achieved with components like Trusted Platform Modules or Hardware Security Modules. We assume the hardware aspects are covered and will be focusing on the software aspects of the architecture.
QA-3: Deployability	The importance of this driver can be understood when one considers that an industrial company may have dozens of plants, each with dozens of edge devices. The manual deployment of production systems will either be infeasible or extremely error prone.
QA-5: Performance (Real-time dashboard)	Whereas this driver seems more directly pertinent to other architectural elements (such as Dashboards), the capability of the IIoT system to deliver data sufficiently quickly is critical for these systems to achieve this goal. Therefore the IIoT architect should aim to achieve latencies much smaller than 15 seconds.
QA-6: Performance (Visual inspection)	Latency in the order of hours could permit some cloud-based and batch-oriented approaches. But when we are dealing with seconds, it becomes evident that the driver will affect an edge architecture decision.
QA-8: Extensibility (New devices and sensors)	The ability to extend devices and sensors from a wide multitude of vendors is a key capability of all Industrial IoT systems.

QA-9: Extensibility (new applications)	This driver is selected for the IIoT iteration because it is cross-cutting. Indeed, it's best for the boundaries of an application to be defined by the use cases and the domain knowledge required for its implementation, rather than the context of its execution (edge device, cloud IoT pipeline, ML pipeline, etc).
QA-10: Observability	This cross-cutting driver will manifest itself in the perceived latency of a system monitoring dashboard, which is not a direct concern of this iteration. However, the dashboard cannot be responsive unless the IIoT backbone is performant. Therefore, this scenario must be considered in this iteration.

Constraints

Architectural driver	Specific IIoT considerations
CON-4: NIST's IoT Cybersecurity Capabilities must be implemented to manage cybersecurity risks	This is a constraint that is specifically focused on Edge, IoT, and Security. See further reading section.

9.3.3.2 Step 3. Choose one or more elements of the system to refine

For this step we are choosing the elements that are highlighted in blue in [Figure 9.4](#) to be refined:

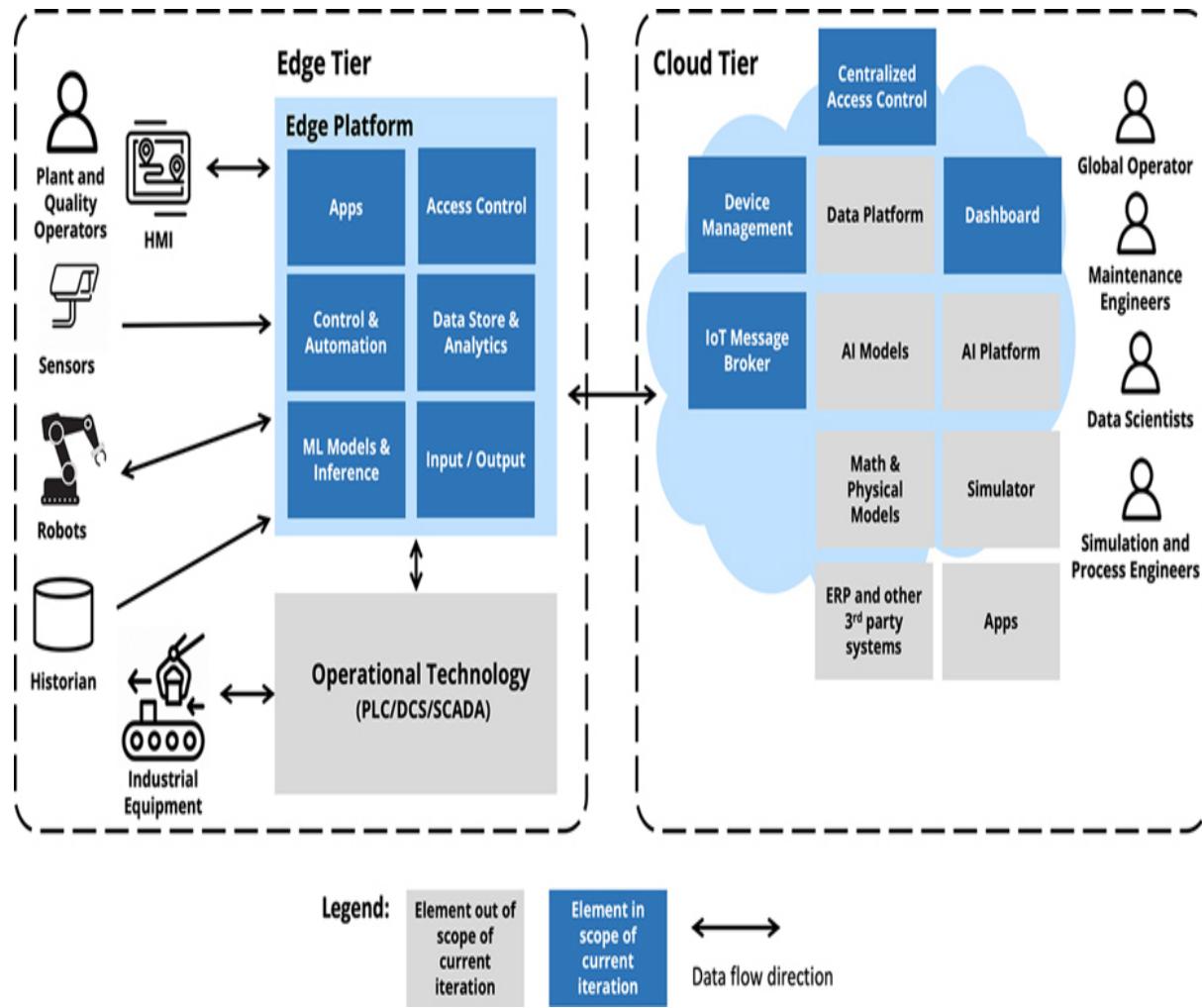


FIGURE 9.4 *Digital Twin Platform reference architecture annotated for Iteration 2*

9.3.3.3 Step 4. Choose one or more design concepts that satisfy the selected drivers

The following design concepts inform the architectural decisions in this iteration.

Design decisions and location	Rationale
Perform computer vision on the edge	<p>This design concept pertains to performing computer vision on a dedicated piece of hardware in close proximity to the production line. The following quality attributes are affected by this decision:</p> <ul style="list-style-type: none"> 1. Performance: By processing the video closer to the source (as opposed to processing it in the cloud), latency is significantly reduced. IIoT applications often require real-time or near real-time analysis and response, so moving the processing closer to the data allows for quicker decision-making. Faster availability of these decisions means faster responses, which enhances operational efficiency and improves overall system performance. 2. Congestion avoidance: High-volume data generated by IIoT devices, especially visual sensors, can strain network bandwidth, if all the data is transmitted to a central location (such as a cloud component) for processing. By performing processing tasks closer to the source one can avoid sending high-volume data over long network segments, only transmitting relevant information. This avoids network congestion which can have adverse effects not only on this data streaming application, but other processes in the plant. 3. Security: video streamed from production lines can contain trade secrets or other sensitive information. If we were to process the data in the cloud, the data stream would become exposed to the public internet and hence many intermediaries. By processing data locally, near the data source, we can reduce the exposure to potential security breaches. 4. Resiliency: in case of an internet link failure, data processing in the cloud will cease, but local data processing can continue. The accumulated data will be much easier to store and transmit to the cloud later, if needed. <p>In addition to these quality attributes, a benefit associated with this design decision is that it should be less costly.</p>

	<p>Alternatives considered: Computer vision in the cloud.</p> <p>Reason for discarding: Many quality attributes, such as performance, would be affected negatively if computer vision would be performed in the cloud.</p>
Adopt the Litmus Edge reference architecture	The Litmus Edge reference architecture, depicted in Figure 9.5, is focused on the design of IIoT and incorporates many design concepts that are useful in the design of Edge solutions.

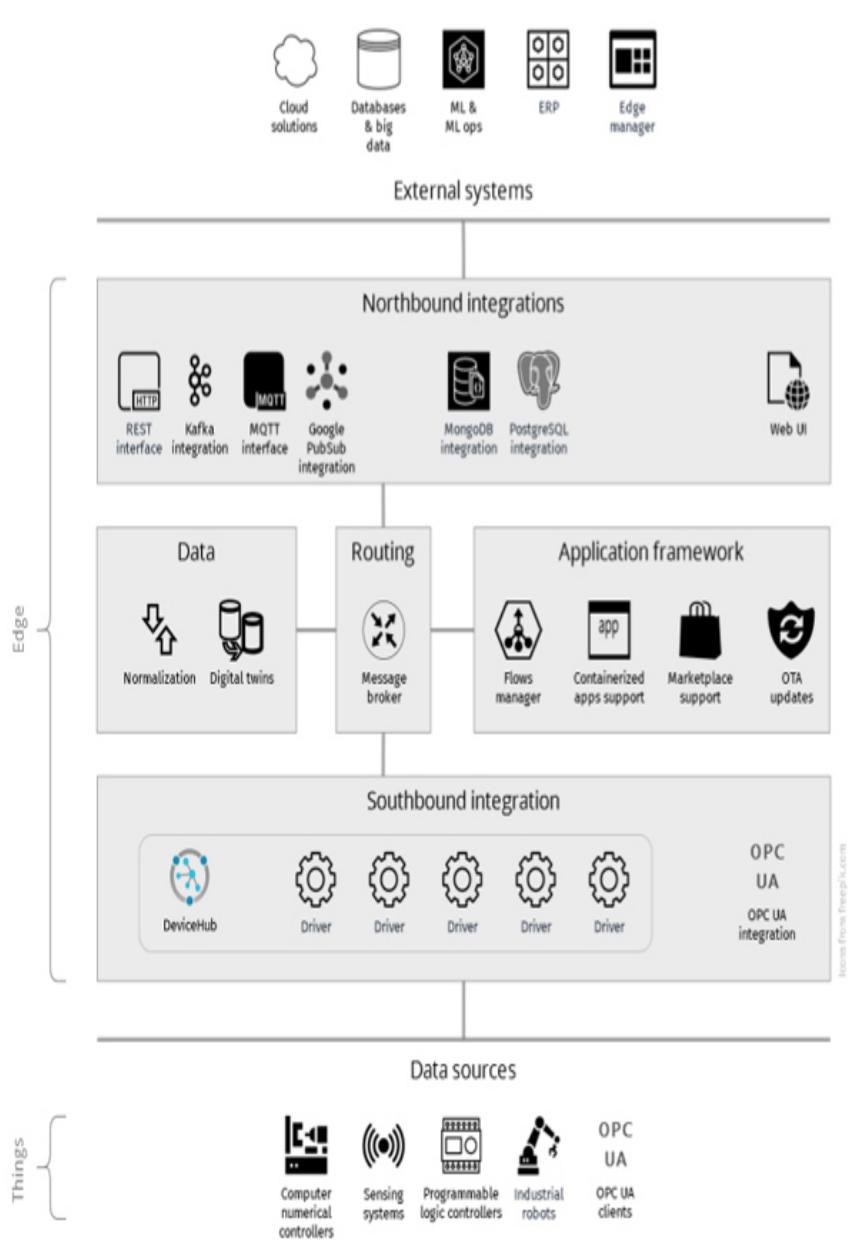


Figure 9.5 *Litmus Edge reference architecture*

Relevant design concepts embodied by the Litmus Edge reference architecture are listed in the following table, and mapped to elements in the diagram.

Design concept	Associated element
<i>Object model for digital twins & meta-modeling—a</i>	Digital twins

	<p>model that can describe the physical world as a collection of interrelated objects with attributes that have a value at certain points in time. Meta-modeling refers to the process of creating a description of the model itself.</p>	(Data)
	<p><i>Connectors</i>—components that are created to reliably link or join elements that do not have prior knowledge about each other. In our case, this provides extensible connectivity between industrial</p>	DeviceHub and drivers (Southbound Integration)

	IoT devices and the Edge platform.	
	<i>Pub-sub architecture</i> —facilitates communication between components in a distributed system, where the actors in the system do not need to know about each other in advance.	Message broker (Routing)
	<i>Pipes and filters</i> —focuses on the flow of data through a series of processing steps. Each filter performs a specific task on the input data and produces an output. The filters are connected together through pipes, which act as the transport for data transfer between filters.	Flows manager (Application s Framework)
	<i>Central edge management</i> —if a client has a vast fleet of IoT devices, located in a large number of factories, they cannot afford to create and maintain deployments on the level of individual edge devices. There is a need for an element that has a "one to many" relation to the edge devices.	Edge manager (External devices)
Alternatives considered: AWS Greengrass reference architecture		
Reason for discarding: AWS Greengrass is a lower-level product—it provides building blocks for creating IoT solutions, whereas Litmus Edge is a ready-made IoT platform. AWS Greengrass would be suitable in projects where a highly customized solution is desired; in our case, the IoT requirements are typical and straightforward, and readily satisfied by an existing generic solution.		

9.3.3.4 Step 5. Instantiate architectural elements, allocate responsibilities and define interfaces

In this iteration, instantiation is performed by choosing technologies to implement the selected design concepts. The instantiation design decisions considered and made are summarized in the following table.

Design decisions and location	Rationale
Establish a contract between "IoT" and "Big Data and AI"	<p>In step 3 of iteration 1, we divided the system into two tiers: cloud and edge. The cloud tier contains elements pertaining to Big Data and AI. In iteration 2 these are treated as black boxes. These elements will be further fleshed out in iteration 3. But before proceeding with architectural decisions it is necessary to establish a contract between the "IoT" and the "Big data and AI" elements.</p> <p>The responsibilities of elements in these elements can overlap. For example, it is possible to do analytics and ML operations using IoT products on the edge, as well as on the cloud. To reduce the ambiguity of responsibilities, it has been decided that:</p> <p>For the implementation of analytic and ML use cases, the "Big data and AI" technology stack will be prioritized, even where relevant capabilities can be developed on the edge.</p> <p>For the implementation of IoT use cases, the "IoT" technology stack will be prioritized, even where relevant capabilities can in principle be developed using cloud services.</p> <p>We further decided that the data flow from the "IoT" elements to the "Big data and AI" elements will be based on the Kafka protocol. This protocol is widely supported and is consistent with the "pub-sub architecture" design concept selected in the previous step.</p>
Use a smart camera to	A smart camera is an edge device capable of real-time inference of

perform computer vision on the edge.

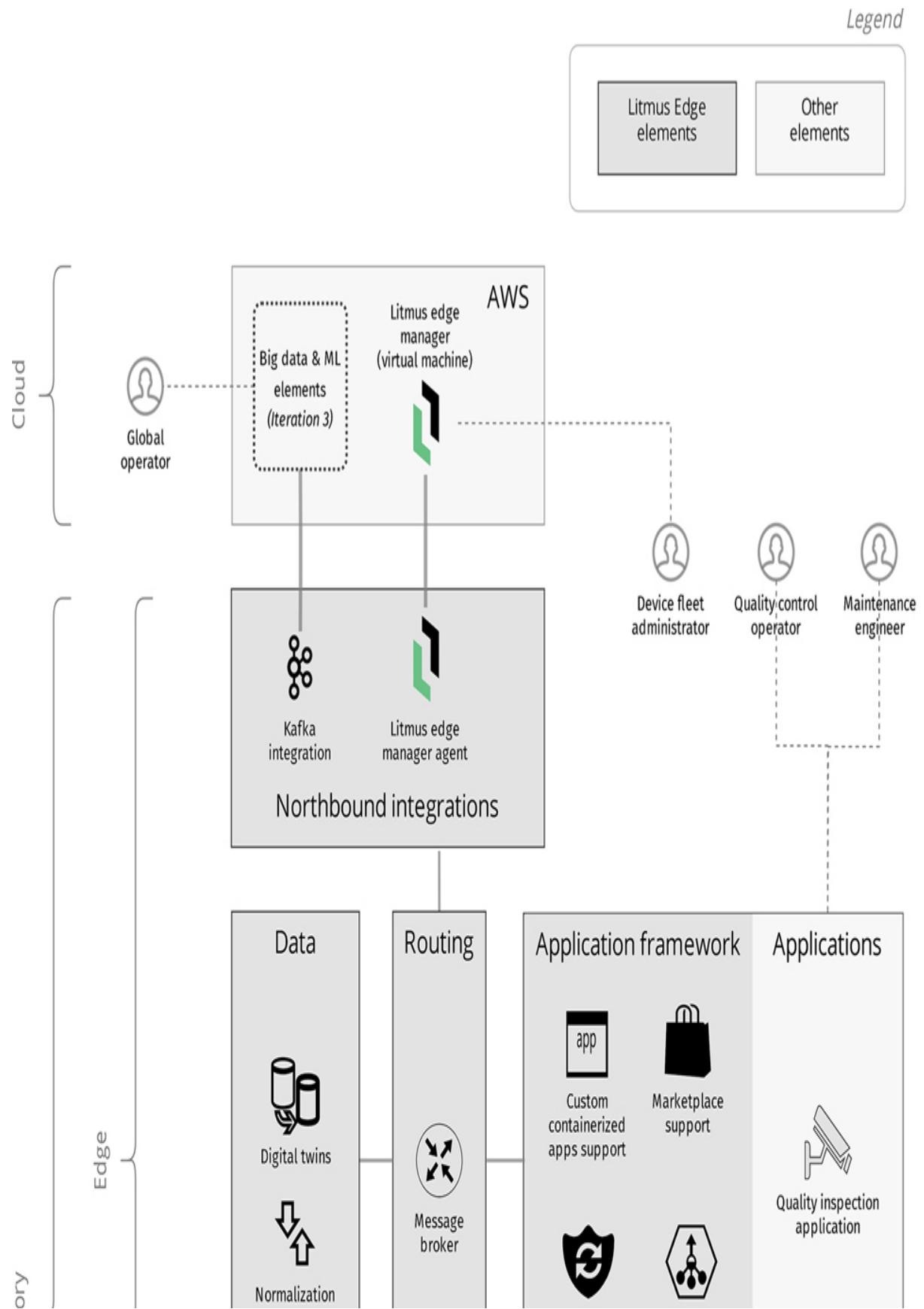
custom computer vision models. The use of a smart camera is the instantiation of the "Computer vision on the edge" design concept. This addresses the requirement of using optical sensing (UC-2, QA-6). Locating the sensor and the computation for ML inference within the same device ensures that the required data bandwidth is available, and reduces the latency between obtaining the image and detecting the defect (for rapid alerting). As the smart camera supports custom ML models, it is possible to train and refine models that are specific to the product being monitored and its common defects.

	<p>The implementation of smart camera solutions is based on NVIDIA Jetson edge-GPU computers, which provides the necessary computational power to enable real-time ML inferencing for scenarios from image classification and object detection to image segmentation. With the open-source NVIDIA TAO Toolkit, these solutions can be developed faster by leveraging pretrained AI models.</p>
Instantiate Litmus Edge reference architecture by using the Litmus Edge platform.	<p>Litmus Edge is a ready-to-use implementation of the Litmus reference architecture, which was selected as a design concept. This decision is compatible with the previous design decision, because Litmus Edge supports custom applications, and thus can host an application that drives the connected smart camera.</p> <p>Litmus Edge is not free. But in this case it is more economical to buy a commercial product than to build the functionality in-house. The client is not in the business of writing software; for them the IoT platform is a means to achieve their business goal, and not a profit center. Therefore they have no incentive to invest in the development of a new system. Additionally, security is hard and expensive to get right. And since security concerns are front and center in IoT, it is usually safer to rely on a battle-tested solution.</p>
Use the Litmus Edge Manager product for the central edge management aspect of the reference architecture.	<p>Litmus Edge Manager is a centralized management platform for edge devices and applications. It is used to set up and manage all aspects of multiple Litmus Edge deployments. This instantiates the "Central edge management" design concept from the reference architecture. Litmus Edge Manager is part of the Litmus product suite and integrates directly with Litmus Edge.</p>



9.3.3.5 Step 6. Sketch views and record design decisions

The diagram in [Figure 9.6](#) shows the result of the prior instantiation design decisions.



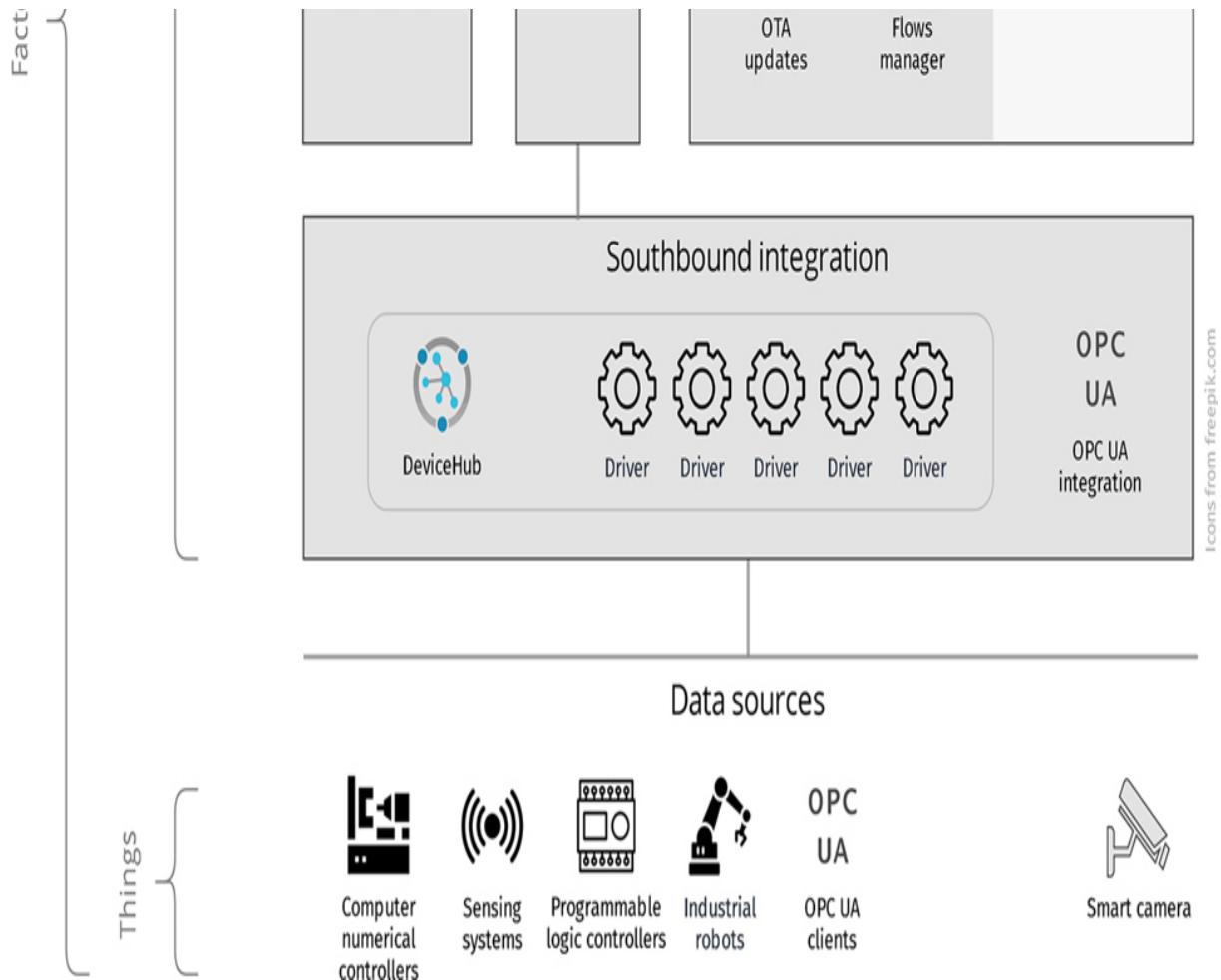


FIGURE 9.6 Instantiation of the *Litmus Edge* reference architecture

In this diagram we highlight a number of elements.

Data sources

The data sources are collections of industrial equipment on the factory floor, referred to as “Things” in [Figure 9.6](#), that generate data and telemetry.

Litmus Edge elements

Element	Corresponding element before refinement (see Figure 9.4)	Responsibility
Southbound integration		
DeviceHub	Input / Output	Allows data connections between devices and Litmus Edge. DeviceHub manages and uses "drivers" that implement connectivity with devices from major manufacturers.
Data		
Digital twins	Data Store & Analytics	Provides a representation of industrial assets, where an industrial asset is an object associated with industrial processes. Includes modeling that allows to assign object attributes, define transformations of input data, and represent a hierarchy of attributes.
Normalization	Input / Output	Normalizes the measurement units and representation of data sent by different devices, ensuring a unified view to data consumers.
Application framework		
Custom containerized app support Marketplace support OTA updates	Apps Control & Automation	Provides the ability to install, update, and launch Docker-based applications from public or private container image registries or marketplaces over the air (OTA).
Flows manager	Apps Input / Output	Graphically defines the data connections and transformations between different components

		of the solution, such as devices, databases, information systems, or static files.
Northbound integrations		
Kafka integration	IoT Message Broker	Sends messages to clients that implement Kafka TCP protocol (to be defined in Iteration 3).
Litmus edge manager agent	Device Management Centralized Access Control	Enables communication with Litmus Edge Manager.

Other elements

Element	Corresponding element(s) before refinement	Responsibility
Applications		
Quality inspection application	Apps Control & automation ML Models & inference	Implements an administrative and operational front end for the Quality control operator. Generates alerts about quality control failure events, per application requirements. Ensures the smart camera updates are compatible with, and can be received by, the cloud applications.
AWS		
Big data & AI elements	See Iteration 3	This is a placeholder that will be instantiated during iteration 3.

9.3.3.6 Step 7. Perform analysis of current design and review iteration goal and achievement of design purpose

The following Kanban table summarizes the design progress and the decisions made during the iteration.

Not Addressed	Partially Addressed	Completely Addressed	Design decisions made during the iteration
	UC-1		<p>Real-time pipeline of data has been organized from the IIoT devices to the cloud.</p> <p>Pending: OEE monitoring will be designed in Iteration 3.</p>
		UC-2	Quality inspection is achieved using a smart camera that performs ML processing on the Edge.
	UC-3 UC-4 UC-5		Out of scope for this iteration
		QA-1	Neither Litmus Edge nor any of the applications affect the critical system functions for the production floor.
	QA-2		<p>Litmus Edge: Authentication, authorization, and RBAC (Role-Based Access Control) policies suitable for enterprise identity management and access control. Access control for all interfaces, including web UI. Quality control application: integration with Litmus Edge security facilities.</p> <p>The Cloud part is pending in the subsequent iteration.</p>

	QA-3		Satisfied by using Litmus Edge + Litmus Edge Manager. The Cloud part is pending in the subsequent iteration.
	QA-5 QA-6		Performance was partially addressed during this iteration, especially QA-6 with the design decision of using a smart camera to perform computer vision on the edge, which decreases latency. Additional performance testing and analysis can be accomplished by creating a trivial skeleton implementation that involves the selected elements (as a proof of concept) and seeing how it fares under the projected load.
	QA-7		Out of scope for this iteration
	QA-8		Satisfied by using Litmus Edge (DeviceHub and device driver library). The Cloud part is pending in the next iteration.
	QA-9		Satisfied by using Litmus Edge (application support). The Cloud part is pending in the next iteration.
	QA-10		Satisfied by using Litmus Edge Manager (admin console dashboard, Digital twins, Grafana dashboards, alerts with multiple event sinks). The Cloud part is pending in the next iteration.
	CON-1 CON-2		Out of scope for this iteration
	CON-3		Out of scope for this iteration

	CON-4		Addressing all of the constraints related to NIST's IoT Cybersecurity Capabilities is deferred to a subsequent iteration.
--	-------	--	---

9.3.4 Iteration 3: Refinement of Big Data and AI Elements

The goal of this iteration is to address Big Data and AI/ML aspects of the system and to select technologies to support system requirements. As usual in ADD, we start by considering architectural drivers in Step 2, to determine how relevant they are for the Big Data and AI aspect of the system.

9.3.4.1 *Step 2. Establish iteration goal by selecting drivers*

All five of the Use Cases introduce requirements for the design of Big Data & AI/ML components. We will review the Use Cases and note what system elements should be considered as we move forward with element decomposition in Step 3.

Use cases

Use Case	Specific Big Data & AI/ML considerations
UC-1: Real-time monitoring and historical data insights	Requires a centralized, cloud-based data repository. Requires streaming data ingestion interface between cloud and plants, streaming data processing in the cloud, and visualization in the cloud.
UC-2: Quality inspection automation	Requires real-time sensor data collection and processing. Requires real-time AI/ML models for anomaly detection. Image processing for optical sensors can introduce channel throughput requirements if done in the cloud; this was one reason for selecting the “computer vision on the edge” option in the previous iteration.
UC-3: Predictive maintenance and XR for maintenance activities	Requires a historical data repository and near-real-time streaming and AI/ML inference capabilities similar to those of UC-1 and UC-2. Inference results should be consumable by extended reality devices, and, thus, should be available via an API access layer.
UC-4: Process simulation	Historical data for raw materials, equipment settings and output quality should be collected and available for manual or ML-based analysis for creation of simulation models.
UC-5: Advanced automation	The results of AI/ML-model inference and simulations should be consumable by automation agents. This may require API data access or the requirement to push messages from the data layer to

components controlling equipment parameters. The control itself (and any ML related decision-making for autonomy) should be executed on the edge device.

Quality attributes

Quality attribute scenario	Specific Big Data & AI/ML considerations
QA-1. Critical system functions for the production floor must be available in the event of a cloud or network outage (applicable to UC-2).	Streaming sensor data collection, processing, and AI/ML anomaly detection models should be running on the edge tier, not in the cloud.
QA-2. An unauthorized access attempt is made. The attempt is denied, logged, and the system administrator is alerted within 15 minutes of the attempted breach. (applicable to all use cases).	<p>Data and AI/ML components should be protected by authentication, authorization, access logging, and access alerting capabilities. Implementing those for every component or centrally is a matter of further tradeoff analysis. On-premise and cloud components might require different implementations.</p> <p>With multiple system components it might be better to introduce a centralized access management system rather than implementing authorization, authentication, and logging in each component.</p>
QA-3. A system deployment is initiated. The deployment is fully automated and supports at least development, test, and production environments. (applicable to all use cases).	Deployment automation should be implemented for all Data and AI/ML components.
QA-4. 100,000 data points (where data point is 120 bytes in average) arrive at the system per second. The system processes them all within one second (applicable to all use cases).	This amount of data should be accounted for in throughput planning for networking components and data processing components. Together with historical data retention requirements it also should be used to define storage capacity.

QA-5. The real-time dashboard for Global Operators is refreshed automatically, at 15 second intervals, with < 15 sec data latency (applicable to UC-1).	Streaming data processing solution design, and near-real-time data repository query response time requirements need to be accommodated. The visualization layer should automatically refresh dashboards.
---	--

<p>QA-6. Industrial processes, coupled with real-time anomaly detection, are continuously visually inspected. Inspection results are received within 5 seconds (applicable to UC-2).</p>	<p>Streaming data processing design for UC-2 (ergo on-prem), including AI/ML inference must be handled. Should clarify why the requirement is “within 5 seconds”; it is not obvious for anomaly detection, as “anomalies” can mean a series of sensor data that is longer than 5 seconds.</p>
<p>QA-7. When a data analyst requests an ad-hoc SQL-like queries for raw and aggregated historical data (multi-relational data that can include various equipment from different plants), 95% of queries return results within 30 seconds (applicable to UC-3,4,5).</p>	<p>Considering the anticipated data volume, this creates response time and scalability requirements for the SQL engine on top of cloud-based data repositories. This also introduces data modeling requirements such as proper partitioning and pre-aggregation, which in turn might require scheduled ETL processing and execution.</p>
<p>QA-8. New devices and sensors can be added to the system at runtime, with no interruption of ongoing data collection and system functionality (applicable to all use cases).</p>	<p>The data model and technologies used for data repositories should be flexible enough to support new devices and new types of devices without data model modification. This might require a metadata repository to store the IoT asset hierarchy.</p>
<p>QA-9. New applications (e.g. Production, Quality, Supply Chain, Sales) can be created building on existing services without the need to re-architect or change the system’s main components (such as Edge, Data and AI Platforms).</p>	<p>No influence to design decisions for Data & AI/ML. Review at Iteration 3 Step 7 to ensure that no limitations were introduced by the design.</p>
<p>QA-10. Key performance metrics and error logs are collected, aggregated, and</p>	<p>This introduces observability requirements to data and AI/ML services. Most likely it</p>

visualized in a monitoring dashboard with a maximum delay of 5 seconds.

should be implemented centrally, with all system components providing metrics/logs in near-real-time. Implementing at the individual component level will most likely be inefficient.

Constraints

Constraint	Specific Big Data & AI/ML considerations
CON-1. Public Cloud (AWS)	Services and technologies used in the solution should be specific to or compatible with the selected public cloud vendor. Particular thought must be given to the integration of the Edge layer and the AWS cloud.
CON-2. Use cloud-native strategy (promote microservices, containers and managed services) for building scalable applications.	Cloud-native services and technologies should take priority when making tradeoffs.
CON-3. Cost economy and cost predictability—specifically in selecting among Cloud service alternatives.	<p>Cost predictability is important for solutions using managed services with a pay-as-you-go (consumption-based) billing model.</p> <p>To satisfy the cost predictability requirement, the solution should explicitly document each component/service cost model and factors influencing it, prioritizing those with less dependency on system workload.</p>
CON-4. NIST's IoT Cybersecurity Capabilities must be implemented to manage cybersecurity risks	Follow the recommendations of the Data Protection section of NIST standard (see Further reading section), which focuses on the ability to encrypt the data.

9.3.4.2 Step 3. Choose one or more elements of the system to refine

Based on the system structure sketch we defined in 9.3.2.5, seven system elements shown in [Figure 9.7](#) are related to Big Data & AI/ML processing and should be refined in Iteration 3.

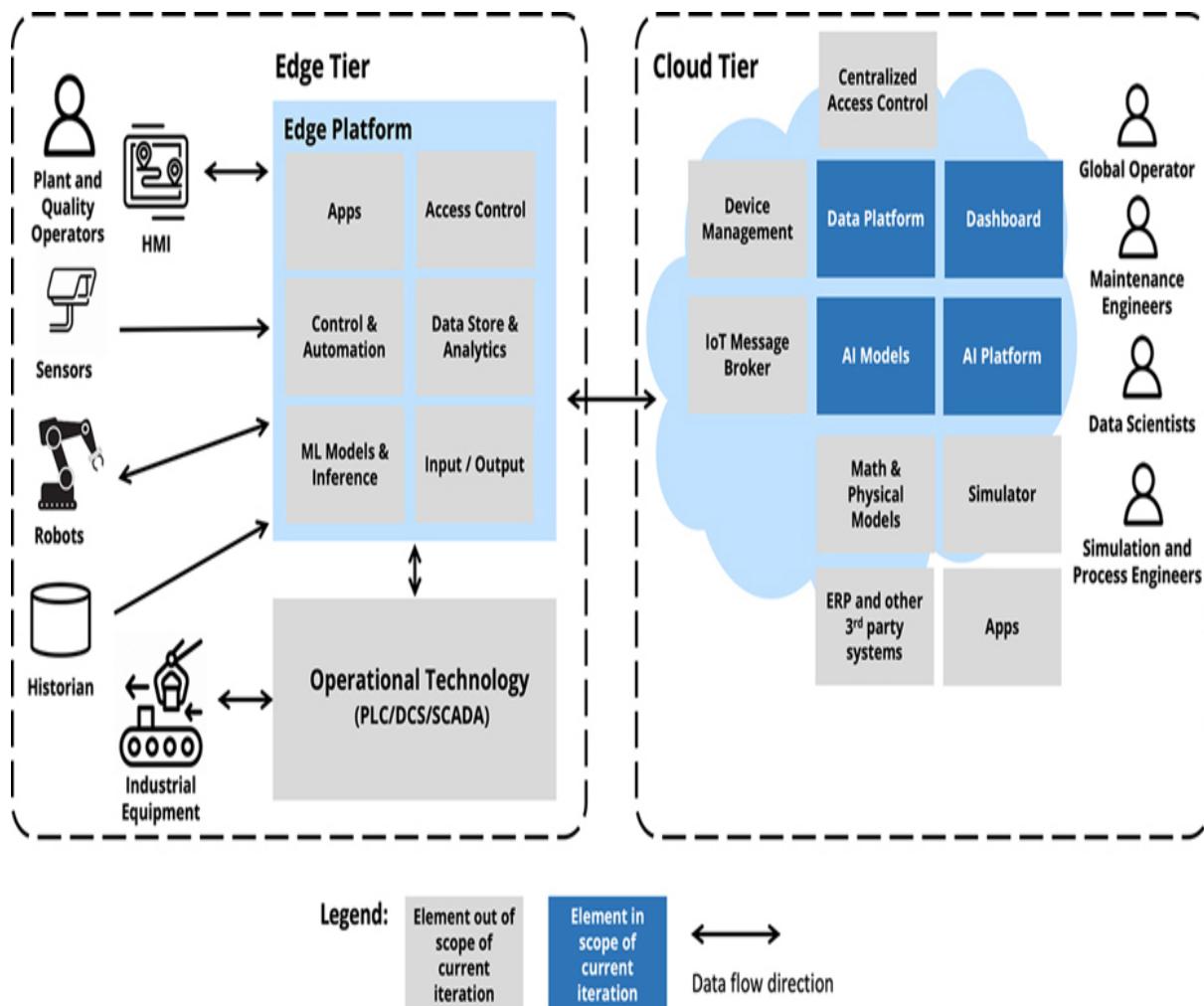


FIGURE 9.7 *Digital Twin Platform reference architecture annotated for Iteration 3*

The Cloud elements to refine are:

- Data Platform
- Dashboard
- AI Models
- AI Platform

9.3.4.3 Step 4. Choose one or more design concepts that satisfy the selected drivers

Design decisions and location	Rationale
<p>Introduce message queues for IoT sensor data transferring between Edge and Cloud layers, as well as between components of the cloud layer.</p>	<p>A message queue provides an asynchronous communication channel between system elements. It simplifies the design by reducing the number of cross-component dependencies, as well as reducing potential performance and scalability issues in synchronous communications. A message queue also acts as a message buffer in case of temporary unavailability of message consumers.</p> <p>The decision to use Litmus Edge made in the previous iteration implies message queue usage on the Edge Layer. Following UC-1 and QA-5, the Global Operator should see the changes from the factory floor within 15 seconds of the event, which means that batching IoT sensor data on the edge and transmitting them to the cloud on a schedule is not a viable option. An alternative of pushing data from the Edge layer directly to some cloud-based data repository is viable, but reduces the options for streaming data processing, which is required for UC-1 and QA-5 as well. Thus, introducing message queues for IoT sensor data at both Edge and Cloud layers is an optimal architectural decision.</p> <p>Discarded alternatives: direct data ingestion from the Edge layer to a data storage service in the cloud, like S3 object storage or one of the AWS database services. Such an approach, however, would introduce undesirable strong coupling between solution components.</p>
<p>Use the streaming data processing pattern to perform data processing operation on individual records.</p>	<p>The near-real-time monitoring and AI/ML inference requirements (UC-1, UC-2, QA-2, QA-5) do not allow for batch data processing. At the same time, the raw data obtained from IoT sensors require additional cleansing, contextualization, and transformation for visual and AI-based analytics. Those calculations should be performed against individual messages from the message queue which makes streaming data processing a good architectural choice.</p> <p>Discarded alternatives: process data in batches according to a predefined schedule.</p>
<p>Separate data serving layers into near-real-</p>	<p>Many implementations of the popular Lambda architecture suggest combining historical and near-real-time data, making this all available</p>

<p>time and historical data layers.</p>	<p>to a visualization layer. This masks the complexity of different data repositories and technologies used for each type of data, simplifying data consumption. However the Lambda architecture comes with implementation complexities and technology choice limitations. In our use case historical (UC-3, UC-4) and near-real-time (UC-1, UC-2) analytics are clearly separated. So, it makes sense to keep these types of analytics separate and allow for different technologies and services (data repositories, data access engines, visualization tools etc.) to be used for each type. Thus, despite both historical and real-time data paths being present, we are not combining their data at the visualization or data virtualization layers.</p> <p>It is worth mentioning that even with this decision we can still satisfy UC-3 (“<i>...predictions on equipment failures and diagnose them by accessing historical and real-time information</i>”). UC-3 can be decomposed into 2 sub-use-cases: using historical data to train AI/ML models and using near-real-time data to get predictions and detect failures. Each of those sub-use-cases can work with its own data time scope.</p>
<p>Use different Edge and Cloud technologies for Data and AI/ML.</p>	<p>Having a minimal number of services and technologies in an architecture is, other things being equal, a desirable goal. It helps avoid duplication of implementation effort, skills acquisition effort, maintenance effort and cost. In addition, CON-2 calls for cloud-native service usage, which requires us to consider technical stack unification based on those services.</p> <p>However, we have two strong arguments <i>against</i> a unified technology stack across Edge and Cloud tier:</p> <ul style="list-style-type: none"> • the set of cloud-native services available in an on-prem environment is extremely limited. In fact, it includes only 3rd-party (often open-source) technologies adopted by cloud providers and implemented as services or cloud-vendor-specific hardware appliances, which are not economically viable for installation across multiple factory environments. • The Litmus technical stack includes multiple services in Data Analytics & AI/ML, including message queues, a time series database, a streaming data processing framework, visualization dashboards, and an ML model runtime environment. Using an alternative technology stack for the Edge Tier would contradict the decision already made in the previous iteration. <p>The decision, therefore, is to separate the two technical stacks—for Data Analytics & AI/ML—utilizing Litmus Edge services on the Edge Tier and following the CON-2 requirement for cloud-native services</p>

for the Cloud Tier.

Discarded alternatives:

- Use a similar set of open-source technologies on the Edge and in the Cloud. While doable, such a decision contradicts CON-2 and would also limit the choice of technologies by having a suboptimal common denominator of Edge and Cloud options.

9.3.4.4 Step 5. Instantiate architectural elements, allocate responsibilities and define interfaces

The first step of the instantiation decisions involve identifying elements associated with the design concepts. The elements identified are summarized in the following table. Once these elements are identified, we are now able to decide which AWS capabilities we will be using to implement them. These decisions, as well as the rationale behind them, are listed in the table below. While technology selection for some elements is predefined by the cloud provider requirement or by decisions made in the previous iteration, some still require additional tradeoff analysis.

Design decisions and location	Rationale
Create a message queue service and implement it with Amazon MSK (Managed Streaming for Apache Kafka)	<p>The message queue service is responsible for data exchange between system components and message buffering, including those between Edge and Cloud Layers. This is an instantiation of the message queue design concept, related to use cases UC-1 and UC-5. It should expose at least an API interface to publishers and subscribers. It should also integrate with the near-real-time data repository and streaming data processing service.</p> <p>Two primary alternatives for message queue service in AWS are Amazon's native Kinesis Data Streams and Amazon MSK. While both would be an adequate choice for this element, only MSK (Kafka) integration is supported by Litmus out of the box.</p>
Create a near real-time data repository and implement it with Amazon Timestream	<p>This is an instantiation of the “No shared data service layer” design concept, and is related to use cases UC-1 and UC-2. It is responsible for storing and serving near-real-time data in the cloud. It should be capable of working with a comparatively short window of the most recent 24 hours of data. It should also integrate with the message queue service and visualization service.</p> <p>The near-real-time data repository contains a relatively low volume of information, limited by the sliding time window. It requires frequent and fast individual record updates. Being used for streaming data processing, it should also exhibit low latency for individual row operations (inserts, updates, row retrieval). The data in the repository will follow a typical time series data pattern, without the need for complex documents or interconnected tables.</p> <p>Such a combination of requirements, combined with CON-2 “Use cloud-native strategy” priority, leaves two major contenders: Amazon DynamoDB and Amazon Timestream.</p>

	<p>While the required data repository can be implemented with both technologies, Amazon Timestream has the advantage of supporting complex SQL queries and native support for time series data analysis functions. This allows us to rely on SQL for aggregations that in DynamoDB would require additional data structures and additional coding to implement.</p> <p>With cost and performance (for the amount of data that fits in-memory storage of Timestream) being comparable, Timestream appears to be a better choice.</p>
Create a streaming data processing service and implement it with Kinesis Data Analytics (Managed Apache Flink)	<p>The streaming data processing service is an instantiation of the “Streaming data processing” design concept and it supports UC-1, UC-2, QA-2, QA-5. It is responsible for message transformations like aggregations and deriving higher-level KPIs from lower-level device data. It should integrate with the message queue service, and should be able to access the asset hierarchy service data.</p> <p>At the time of writing, the choice of technologies for streaming data processing (ignoring proprietary solutions) is between Apache Spark, Apache Flink, and Apache Beam. CON-2 (managed service usage) and CON-1 (AWS) narrows this choice down to the first two, which are represented in AWS by two managed services: AWS Glue (can leverage Apache Spark) and Kinesis Data Analytics.</p> <p>With closely matched capabilities, the choice between Spark and Flink is often decided by the skills and preferences of the development team. Having no such preferences documented, we will go with Apache Flink as the more streaming-oriented processing framework.</p>
Create a visualization service and implement it with Amazon Quicksight	<p>The visualization service is not directly an implementation of the design concepts previously identified, however, it is necessary to support UC-1, UC-4, QA-5. This service is responsible for data visualization for monitoring and analysis purposes. It should integrate with the near-real-time data repository and historical data repository.</p> <p>The choice of visualization service technology is mostly predefined by CON-1 (AWS as a cloud provider) and CON-2 (managed service usage). In the absence of specific requirements on visualization types, security model,</p>

interactive capabilities etc., the native QuickSight service is a safe choice that can be revised later if needed.

<p>Create a historical data repository and implement it with Amazon Athena and S3</p>	<p>The historical data repository is also an instantiation of the “No shared data service layer” design concept that supports QA-4 and QA-7. This repository is responsible for storing and serving historical data. It should be capable of working with large amounts of data, reflecting multiple years of system operations. It should integrate with <i>visualization service</i> and <i>AI platform service</i> and also provide ad-hoc data querying capabilities to <i>end users</i>.</p> <p>The two alternatives for large analytics data repositories are Amazon Redshift (a columnar MPP database) and Amazon Athena (AWS’s version of Apache Presto SQL engine, running on top of S3 data and additional services like the Glue Data Catalog).</p> <p>The basic use case for Amazon Redshift is a data-warehouse-like solution, with a well-defined data model and a set of predefined reports running on top of it. The use case for Amazon Athena is a data-lake-like solution, with ad-hoc queries on top of raw or slightly preprocessed data.</p> <p>As the second use case is closer to the patterns of historical data processing defined by UC-1, QA-7, and the need to train ML models, we will go with Amazon Athena.</p>
<p>Create an asset hierarchy service and implement it with Amazon DynamoDB</p>	<p>This service is an instantiation of the “Different technologies on the edge and in the cloud” design concept which directly supports QA-8. The service is responsible for storing, managing and serving metadata on IoT devices, from individual sensors to machines to production lines to factory level. Unlike the Edge service of the same name, it will be used for managing only the higher levels of hierarchy, starting from the <i>factory</i> and up. It should consume and replicate lower levels from the Edge instances, provide capabilities to define and change the hierarchy to <i>end users</i>, and provide a data access interface to <i>streaming data processing service</i>.</p> <p>The asset hierarchy is a comparatively small dataset with a straightforward data model and without specific performance requirements or supported data access patterns. As such, a wide variety of technologies can be used to implement it. The selection of Amazon DynamoDB here is due to the fact that this database is used for asset hierarchy by Amazon’s IMC (Industrial Machine Connectivity) framework, which can be partially reused for our</p>

	solution as well.
Create an AI platform service and implement it with Amazon Sagemaker	<p>This service is also an instantiation of the “Different technologies on the edge and in the cloud” design concept that supports UC-2, UC-3. It is responsible for AI/ML model development, training, quality control, evolution, and execution. It should provide capabilities to develop and evolve models to <i>end users</i>, have access to a historical <i>data repository</i>, and provide capabilities to export developed models for deployment on the Edge.</p> <p>Amazon Sagemaker is chosen as it provides a rich set of services for ML model development.</p>
Use Generative AI for maintenance and diagnostics (UC-3) with Amazon Bedrock	<p>Recently, a new service appeared on the AWS platform based on Generative AI: Amazon Bedrock. After a thorough analysis of UC-3, the team conceived the idea of implementing an AI-powered Co-Pilot for maintenance engineers. The Co-Pilot's purpose is to process equipment technical documentation and provide answers to user questions in natural language.</p> <p>It was decided to integrate the Co-Pilot into the XR application for field maintenance by leveraging speech-to-text headset functionality, thereby saving time on searching and reading documentation. AWS Bedrock enables this “Talk to documents” scenario by providing a Large Language Model (LLM) as a fully managed service. The user’s query, submitted as text, is received via an API. The corresponding equipment documentation is then retrieved and processed by the LLM to provide a reply that addresses the user’s question</p> <p>The previous decision to connect the XR device with the Cloud, bypassing the Edge, allows this scenario to be implemented with limited modifications.</p>

9.3.4.5 Step 6. Sketch views and record design decisions

Figure 9.8 presents a simplified data flow diagram to show the results of the instantiation step. The cloud tier also includes the cloud resources used in each of the elements.

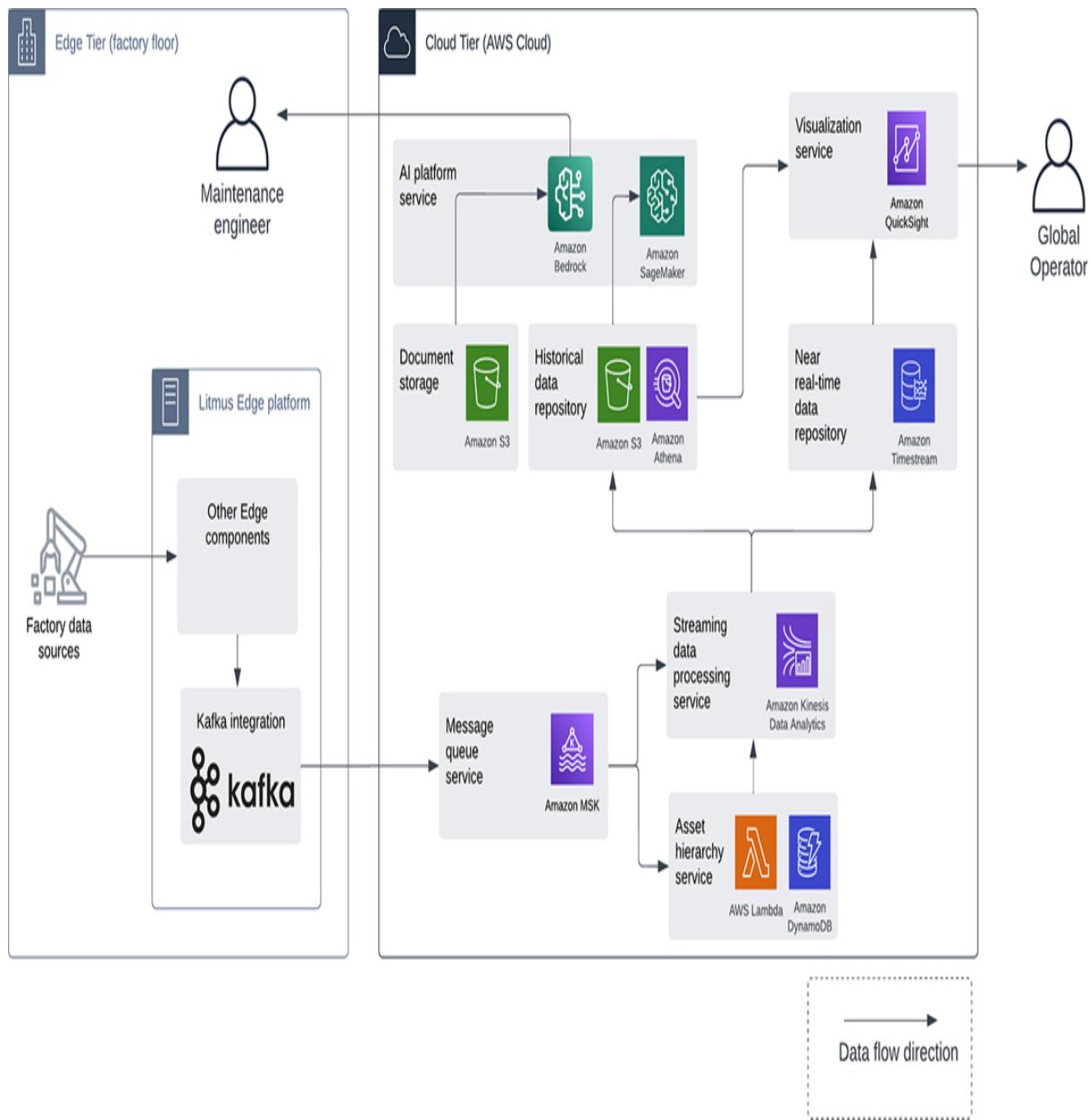


FIGURE 9.8 Simplified data flow diagram for the Digital Twins system

The data flow at the Cloud layer starts with Kafka integration passing the data collected on the Edge to the Message Queue Service. The messages are then picked up by the Asset Hierarchy Service (to build and store equipment inventory from individual messages) and by the Streaming Data Processing Service for cleansing,

enrichment, aggregation and other processing purposes. Next, the data is stored in two separate repositories for near-real-time and historical analysis. Finally, the data is used by the AI Platform Service for AI/ML model development and evolution, as well as by the Visualization Service to present the data to end users.

It is worth mentioning that the diagram here simplifies the real data flow for the sake of readability and architecture clarity. For example, in practice the flow between the Message Queue Service and the Streaming Data Processing Service is almost always bi-directional: raw data is being processed and then put to the message queue again for the next layer of the processing pipeline to pick up.

[Figure 9.9](#) provides a mockup of a real-time dashboard for UC-1.

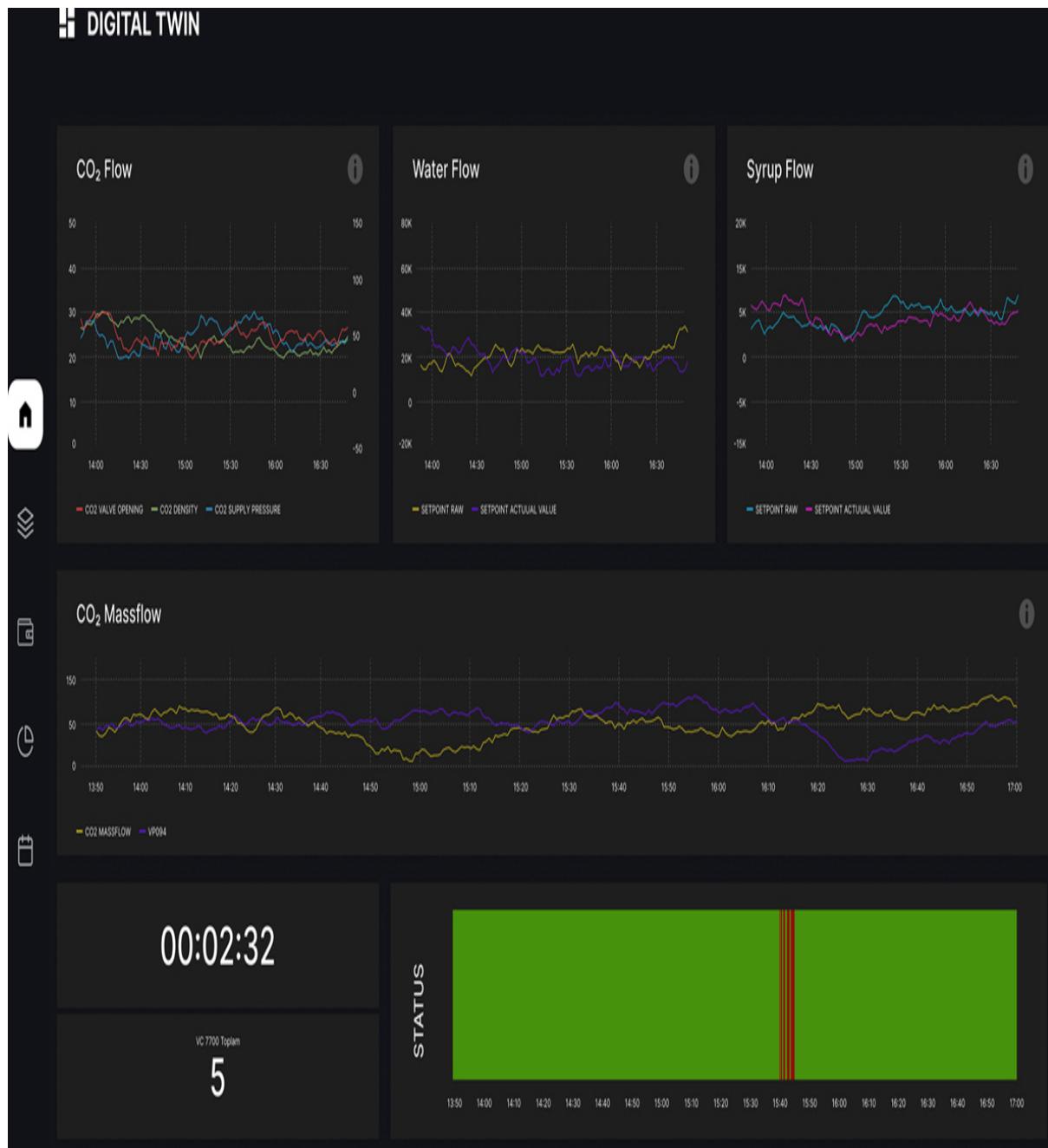


FIGURE 9.9 Real-time Dashboard Mockup

A mockup of the Extended Reality (XR) user interface for the maintenance activities described in UC-3 can be found in [Figure 9.10](#):



FIGURE 9.10 The Extended Reality (XR) user interface for maintenance activities

9.3.4.6 Step 7. Perform analysis of current design and review iteration goal and achievement of design purpose

As in the previous iteration, we now can summarize the design progress and review the addressed architecture drivers.

Not Addressed	Partially Addressed	Completely Addressed	Design decisions made during the iteration
		UC-1	All the related elements are established and associated with cloud provider services.
	UC-2 UC-3		While the design decisions provided the environment for AI/ML model development, this iteration did not cover AI/ML algorithm selection, which needs to be addressed in future iterations.
	UC-4 UC-5		Out of scope for this iteration
		QA-1	Covered in the previous iteration
		QA-2	While not discussed explicitly, data access control is present in all the selected services and technologies. It is

			also integrated with AWS IAM (Identity & Access Management service) and Cloud Watch (logging and monitoring service).
	QA-3		It is left for future iterations to make a selection between Amazon CloudFormation and Terraform as two primary IaaC approaches in AWS, as well as to design CI/CD pipeline for the whole solution.
		QA-4	Both Amazon Timestream and Amazon Athena with S3 storage are capable of storing and processing this amount of data. Selected data processing services scale horizontally and will be able to manage this amount of data as well.
	QA-5		While Amazon QuickSight itself does not provide automatic dashboard refresh capabilities, it is often used for embedding reports into custom web interfaces. If that is the case, such a custom interface is also able to control the dashboard refresh frequency.
	QA-6		Out of scope for this iteration
		QA-7	Amazon Athena provides a SQL interface to query historical data.
	QA-8		Partially addressed by the introduction of the Asset Hierarchy architectural element. This should be further developed to define specific data models and Edge messages used to extract equipment information.
		QA-9	Satisfied by the decisions made in this iteration (such as the selected AWS services) as well as the decision from

			the previous iteration (Litmus Edge).
	QA-10		Addressed by inherent AWS capabilities and services integrability with Amazon CloudWatch as well as the decision from the previous iteration (Litmus Edge Manager).
	CON-1		Fully addressed by using AWS as cloud provider of choice.
	CON-2		All the specified cloud services are managed services.
	CON-3		This is only partially addressed as a number of managed services were chosen in this iteration. A cost model based on workload parameters should be introduced later to address CON-3. This may result in some changes to the technologies that were selected.
CON-4			Out of scope for this iteration

9.4 Summary

In this chapter, we explored an example of using ADD in the emerging realm of Digital Twins within an Industry 4.0 context. The complex nature of this domain blends areas like IoT, cloud computing, big data, AI/ML, XR, simulation, advanced automation, and robotics, presenting a multifaceted challenge. Such a broad spectrum surpasses the capabilities of any single architect. Each of the iterations was led by a different architect, but their decisions needed to be carefully aligned. After the first iteration, which was focused on the overall structuring of the system, iterations 2

and 3 were performed by architects with different backgrounds, who made an analysis of considerations for each chosen driver. This was something we did not see in the previous case study, but it was deemed to be essential to managing complexity and aligning decisions between the iterations.

The decisions made in this chapter were shaped by the diverse design concepts, patterns, and technologies from each discipline. The rapidly evolving nature of this domain means that established information sources, such as reference architectures or vendor documentation, could not possibly offer direct guidance for satisfying all architectural drivers. The architects needed to perform experiments and build prototypes to arrive at their architectural decisions.

Digital transformation programs, as described in this case study, are typically multi-year journeys. The architects involved should be prepared to periodically review their decisions, as new technologies or updated versions are bound to emerge during the implementation, presenting promising business and architectural opportunities. This case illustrates how, within a relatively short period of time, architects can witness the emergence of new technological capabilities, necessitating a proactive approach to continuously integrate technological advancements into their designs.

9.5 Further Reading

You can read about Industry 4.0 here:

(<https://en.acatech.de/publication/industrie-4-0-maturity-index-update-2020/>)

A short essay entitled “The Rise of Digital Twins: How Software is Eating the Real Economy” discusses digital twins and their many use cases. This can be accessed at:

<https://info.softserveinc.com/hubfs/files/the-rise-of-digital-twins.pdf>)

More information about the Litmus Edge IoT platform can be found at: <https://litmus.io/litmus-edge/>

The AWS Architecture blog includes a very interesting “Manufacturing” category:

<https://aws.amazon.com/blogs/architecture/category/industries/manufacturing/>

The NIST standard describes IoT Cybersecurity Capabilities including recommendations for Data Protection. This can be found at: <https://pages.nist.gov/IoT-Device-Cybersecurity-Requirement-Catalogs/>

A discussion of how and when to prototype as part of the architecture design process can be found in: H-M Chen, R. Kazman, S. Haziye, “Strategic Prototyping for Developing Big Data Systems”, *IEEE Software*, 2016.

Finally, a design concepts catalog that includes some of the reference architectures and technologies mentioned in this case study is part of the Smart Decisions Game, which is described in H. Cervantes, S. Haziye, O. Hrytsay, R. Kazman, “Smart Decisions: An Architectural Design Game”, *Proceedings of the International Conference on Software Engineering (ICSE) 2016*, (Austin, TX), May 2016. More information can be found at: <http://smartdecisionsgame.com>

10. Technical Debt in Architectural Design

In this chapter we are focused on design debt and particularly in cases where that design debt is unintentional debt. We will look at the ways in which properly thought-out architectural design decisions can help to avoid or remediate this debt. We start with an introduction to technical debt, followed by a discussion of its roots and how debt can be addressed through refactoring and redesign. We also discuss how ADD can be used to design with technical debt in mind.

Why read this chapter?

You have almost certainly heard about technical debt and you may already have some notion of what it is and why it is bad. But there is little discussion of how to design to avoid debt, or how to redesign, via refactoring, to reduce debt. In this chapter you will learn about design debt and how it can be managed using ADD.

10.1 Technical Debt

Technical debt is the often unintended burden of complexity that every software project accumulates over time. We do not intend to add debt to a system as we develop, maintain, debug, and extend it, but it happens just the same. It happens because software is complex, with lots of moving parts, because dependencies among the parts are often indirect and invisible, and because our understanding of this complexity is limited in scope, and imperfect. Sometimes we (again, unintentionally) add to this complexity because we are focused on our task at hand, fixing a bug or implementing a feature and don't realize that our efforts are decreasing the conceptual integrity of the system. Perhaps we implement "nearly but not quite the same" functionality as what already exists in some other part of the system. Perhaps we make a direct dependency between two parts of the code base without realizing that we should have employed an abstraction, via a published API. These kinds of degradations of an architecture happen all the time, typically without developers being aware they have contributed to the system's technical debt.

Other times we *consciously* choose to add technical debt—doing some expeditious hack in pursuit of meeting a deadline. And perhaps we do this with the intention of going back later and cleaning it up and "paying back" the debt. This is a completely reasonable strategy, particularly for startups or exploratory prototypes or when you are faced with a hard deadline and missing that deadline has serious business consequences.

You can acquire many different kinds of technical debt in a complex software project—code debt, documentation debt, deployment debt, testing debt, and so on.

10.2 The Roots of Technical Debt in Design

Design debt—that is technical debt in design—can have its roots in many sources. It can originate from drivers that are not well defined, particularly quality attributes. It can emerge as a by-product of the pressure to deliver quickly, so not much of the project schedule is devoted to design. In such cases developers feel like they are making more progress by delivering features, irrespective of the quality of the underlying architecture. This attitude is reinforced in most projects because backlogs typically only contain features and bugs, but not "enablers" of architecture.

Even when design is a priority, unintentional debt can accumulate because the design process is complex: there is an abundance of design concepts and making a good choice among them may be non-obvious and require a great deal of analysis and prototyping (which is frequently omitted as it may seem to "get in the way" of progress). And even when the design concepts are chosen wisely, there may be a lack of understanding of the architecture by developers, resulting in instantiation and implementation mistakes that undermine the architectural vision. For example, we have seen many cases where the "as implemented" architecture diverges wildly from the "as designed" architecture.

And design is not always done well. A survey of 1831 software practitioners (developers and architects) done by the Software Engineering Institute found that bad architectural choices were the greatest, most problematic sources of debt in their projects. Furthermore, as a system evolves there are many ways in which design debt can be introduced and can accumulate. These all involve inadequate attention to coupling and cohesion, in some form.

- “Improper separation of concerns”, where multiple responsibilities are implemented in the same class or module, making the class or module bloated, difficult to understand, test, and modify.
- “Clone and own” where a piece of code is copied (cloned) and then modified, typically because the programmer does not understand the dependencies on the original code and so fears to modify it or take time to understand it.
- “Tangled dependencies”, sometimes called “big ball of mud”, where there has been little thought given to architectural integrity, and dependencies are added (unintentionally) over time, eventually resulting in unmanageable complexity.
- “Unplanned evolution”, which is where new features and bug-fixes are added without any consideration of how these changes affect the overall structure and conceptual integrity of the system. This is rather common today in many teams following an Agile methodology where effort is devoted to addressing the immediate, pressing issue—implementing a customer-facing feature or fixing a bug. While this customer focus makes perfect sense in the short run, developers typically pay little attention to the impact of such changes in terms of the maintainability, understandability, or modifiability of the code base over time.

This list is not exhaustive, but these are some of the most common causes of design debt in the maintenance phase of a system. Now you might ask, why do these practices lead to debt, and how do we know this? Quite simply, they lead to debt because they violate the principles of good design: these practices tend to increase coupling or decrease cohesion. They degrade the modular structure of a system,

which makes the system harder to understand and modify, and since systems continuously evolve this degradation of modifiability is highly problematic.

To understand why this is the case, let's take a quick look at the SOLID principles. The SOLID principles have been around for over two decades and have come to represent the highest ideals of good design. The acronym derives from the first letter of each principle: Single-responsibility, Open-closed, Liskov substitution, Interface segregation, and Dependency inversion.

These principles were originally devised with a focus on object-oriented programming, but in fact they transcend any particular technology or design approach. Let's look at these principles in terms of their implications and their consequences:

- The *Single-responsibility principle* says that there should never be more than one reason for a module to change. This means that a module should have high cohesion.
- The *Open-closed principle* says that software entities should be open for extension, but closed for modification. A module is “open” if it can be extended, for example by inheritance and this means that the original module is cohesive, doing just one thing. And for a module to be closed for modification it must expose a well-defined interface, which limits coupling opportunities.
- The *Liskov substitution principle* says that functions that use references to base classes must be able to use objects of derived classes without knowing it. This again means that a function can only depend on the class’s interface, which limits coupling.

- The *Interface segregation principle* says that clients should not be forced to depend upon interfaces that they do not use. This implies that large, complex interfaces should be split up into smaller, more focused ones. This is management of cohesion.
- The *Dependency inversion principle* states that code should depend upon abstractions, and not concretions. Following this principle naturally leads one to design software with low coupling.

In each case the five principles that encompass SOLID are specific ways to think about and to achieve low coupling and high cohesion. David Parnas wrote about this over 50 years ago in his seminal paper “On the Criteria to Be Used in Decomposing Systems into Modules”. He was talking about coupling and cohesion back then, although he did not use those terms. Five decades later, the software industry is still struggling to achieve these goals. Why is this?

There are many possible answers to this question, but we boil it down to two major issues: incentives and lack of measurements.

Let us consider a similar question to put this into perspective. Why is the population of the United States of America the most obese out of all the world’s major economies? Nearly 70 percent of Americans are obese or overweight. And why, according to the Centers for Disease Control and Prevention, does it keep getting more overweight with each passing year? This epidemic of obesity costs many billions of dollars each year. Our claim is that the real problem is incentives. Quite simply, we humans love our cakes and our ice cream and our sweet drinks. Sweets trigger a dopamine reaction in our brain. We can quite easily become addicted to them. We are genetically incentivized to seek out sweets. And while we can control

such responses—our genetics do not determine our destiny—without good feedback and direction, we are unlikely to do so. Our nature is to seek them out.

What is the connection with software development?

Developers are measured by outcomes such as number of features shipped or bugs fixed in a unit of time, number of commits or lines of code committed per unit time, or more global measures such as cycle or sprint time. Each of these outcomes—fixing a bug, making a commit, implementing a feature—is a short-term behavior that is rapidly rewarded; it is the software equivalent of a shot of dopamine. Why wouldn't we want more of this stuff? Some organizations track each developer's software velocity and reward high achievers and penalize low achievers. Naturally people will respond to such incentives.

Furthermore, humans respond to short-term, immediate gratification over longer term broader goals. We all know we should eat better and get more exercise. But the couch looks comfy and those cookies are yummy. Cookies and the couch win more of the time, sadly.

In fact, the situation is even worse in software development than in our analogy with human health. At least with our own bodies we can see ourselves in the mirror, we can stand on the scale and see if the numbers have gone up, we can feel our clothes becoming too tight, we can acknowledge that we are out of breath after climbing a flight of stairs. What are the equivalents in software development that might give us similar feedback? Yes, some organizations—rare ones—do measure their software development velocity as we just said, and this is great. It is a start. These organizations can use this data to incentivize (and punish) developers, but they could also use such data to view and track the accumulated consequences of their *bad* software development habits. But even those

organizations that do track their productivity seldom understand exactly how they have gotten into this mess and even less often have a concrete plan for getting back into shape.

While organizations may measure the *consequences* of their technical debt, they are not measuring the underlying *causes* of their problems: their increased coupling and decreased cohesion, or the lack of time or knowledge to make adequate design decisions and good architectural choices. And these are design issues. They occur due to a lack of attention to design, perhaps when an architecture is first conceived, or perhaps as the architecture evolves. But technical debt is, to a large extent, a design problem. How do we deal with this problem? First, it needs to be recognized up front as a design driver. If so—if we have collected some maintainability or modifiability scenarios—or if it is recognized as a concern by the architect, then we can justifiably design an architecture to have low coupling and high cohesion, or we can provide adequate support for making appropriate design decisions. We do this through the application of appropriate patterns and tactics and by following a systematic design process such as ADD, with enough time to perform it adequately.

Second, to maintain an architecture with low coupling and high cohesion—that is, one with a lower probability of accumulating technical debt—we need to monitor it for signs of debt and take appropriate action when the architecture shows signs of deterioration. You can think of this as your annual checkup with the doctor. The doctor may take your blood pressure, do blood tests, listen to your heart and so forth. Each of these tests is done to determine the early warning signs of a potential problem. These tests are done so that appropriate remedial measures can be adopted. In the case of our bodies, the appropriate remedial

measures might be diet or exercise, or cholesterol-lowering medication. For our software systems the appropriate measures are typically refactoring and redesign.

10.3 Refactoring and Redesign

Kruchten and colleagues have written that a project's backlog—and hence the investments and effort that will be allocated to software in the future—may have positive or negative value, and may be visible or invisible. This is depicted in [Figure 10.1](#). If some part of the software has positive value and is visible, we call this a feature. If it has negative value and is visible, we call this a bug. So far so good. But the invisible parts of software are more tricky. If the software has positive value and it is invisible, that is architecture and infrastructure: We need it, but no end user is ever aware of it. It is simply part of the cost of doing business. And if the software is both invisible and has negative value, that is technical debt. Everyone understands the difference between bugs and features, and there is little discussion that we should minimize the former and maximize the latter.

The tension between architecture and technical debt is, on the other hand, more subtle and fraught. Why is this? To examine this question, let us return to the discussion above about incentives.

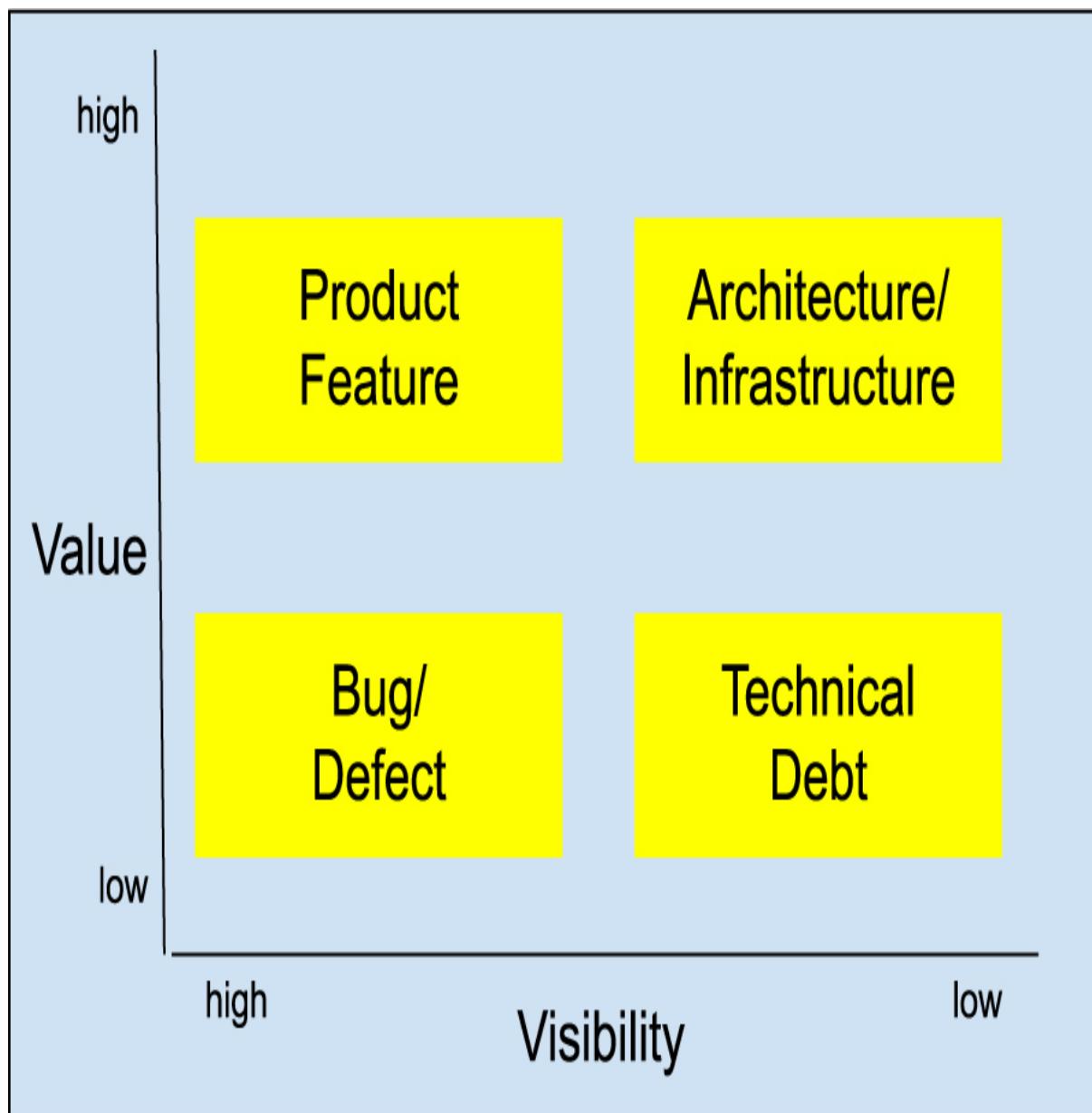


FIGURE 10.1 *Value and Visibility: The Dimensions of Entries in Your Project Backlog*

All software systems accumulate debt over time. This is one of Lehman's Laws of Software Evolution: "as [a system] evolves, its complexity increases unless work is done to maintain or reduce it". Systems always trend towards disorder—you can think of this as a kind of "entropy". This can, however, be counteracted, as Lehman pointed out. To counteract architectural degradation we need to refactor, to

restore order (see the sidebar “Refactoring”). This refactoring can take many forms: removing unwise dependencies, reorganizing functionality to make it more cohesive, splitting large modules into smaller ones, creating abstractions with published APIs, changing a technology that was previously selected incorrectly, and so forth. Each of these refactorings comes at a cost of course.

Refactoring

If you refactor a software architecture (or part of one), what you are doing is maintaining the same functionality but changing some quality attribute that you care about. Architects often choose to refactor because a portion of the system is difficult to understand, debug, and maintain. Alternatively, they may refactor because part of the system is slow, or prone to failure, or insecure.

The goal of the refactoring in each case is not to change the functionality, but rather to change the quality attribute response. (Of course, additions to functionality are sometimes lumped together with a refactoring exercise, but that is not the core *intent* of the refactoring.)

Clearly, if we can maintain the same functionality but change the architecture to achieve different quality attribute responses, these requirement types are orthogonal to each other—that is, they can vary independently.

Take a hypothetical project that is humming along, implementing features and swatting bugs. Perhaps the development velocity per developer is going down, but the project manager can often combat this with hiring or better testing or techniques such as code reviews. This manager is almost certainly incentivized to produce more features, and quickly. Now suppose that an enlightened developer looks at

the project's history, notes that the feature velocity (per developer) has gone down and bug rates have gone up and bug resolution time has gone up. So she comes to her manager and offers to "pay down" some of the accumulated debt via refactoring. How does her manager react? Her manager has just been given a proposal to spend precious developer resources on the implementation of zero new features. Most managers, faced with this proposal, will say "Hell no! Get back to work, implementing features and swatting bugs!". Why? Again, because of incentives and the lack of appropriate measures. Few managers are incentivized to have a cleaner, more coherent, better organized code base.

If, on the other hand, the developer had been able to present a business case to her manager, showing the return on investment of the refactoring, perhaps then the discussion would have gone differently. If she had been able to say: "with this investment of 3 person months of effort we will increase our development velocity by 30%, and achieve a 300% ROI in the first year alone" she might have gotten the green light. Few projects actually have this discipline, and few projects collect the data necessary to make such an assessment. But they should. And there are existing tools to help make this business case, such as DV8 (see the Further Reading).

Let us now turn our attention to the relationship between technical debt and ADD. In essence we need to ask ourselves: how can we design to "future proof" our systems as much as possible, so that they accumulate less debt. And, as it turns out, ADD can help.

10.4 Technical Debt and ADD

The ADD method is not aimed at any single quality attribute or concern; however, its activities can be easily focused on specific concerns. Here we walk through some of the steps of ADD and discuss how these steps can be adapted so that they specifically address technical debt concerns. In what follows we will focus on the quality attribute of modifiability, but these techniques can be applied to other quality attributes as well.

Step 1:Review inputs

In this step we need to make sure that one of the concerns included in the inputs to the design process is to avoid debt. In fact, we believe that this should be a concern in almost every complex software project. We will achieve this in our design process, as we will see, by making design decisions carefully. For example ensuring that we choose tactics and patterns such that coupling is minimized and cohesion is maximized. These are necessary but not sufficient conditions. If a beautiful design is created but developers do not follow it, if the design is not adequately communicated and socialized, if the design is undermined by subsequent maintenance activities, or if design decisions are not documented then any goodness baked into the original design will be undermined.

For example, in the Hotel Pricing System, a future requirement involves doing analytics with respect to price changes. It is critical that this evolutionary scenario is collected as part of this step.

Step 2:Establish iteration goal

In this step drivers are chosen that will be the focus of the current iteration. An architect who is concerned about technical debt should, at this point,

elicit or create modifiability scenarios that characterize the ways in which a system is expected to evolve. When eliciting or generating scenarios we should always consider: 1) expected uses of the system, 2) expected, planned growth cases for the system, and 3) exploratory scenarios, where more speculative aspects of the system's evolution can be considered. While the future state of the system can never be fully predicted, the mere consideration of such scenarios will steer the design process in the right direction. To put it another way, without the consideration of such possible states, it is highly unlikely that design decisions will be put into place to "future-proof" the system. For example, in the Hotel Pricing System, an evolutionary scenario that focuses on the introduction of an analytics component that helps users in changing prices, may be created.

ID	Quality Attribute	Scenario	Associated use case
QA-10	Modifiability	A component that performs different types of analyses on price changes is introduced to the system. The component is added successfully without requiring changes in the code or interruptions in operation.	HPS-2

Steps 3-5:Choose and instantiate elements

In choosing and instantiating architectural elements, the wise architect will consider modifiability tactics and patterns (see [3.5.2](#)). One of the most common ways that technical debt takes root in a complex code base is through unbridled dependencies. Most of the modifiability tactics exist to combat this—to

reduce coupling and to increase cohesion. An example of a modifiability tactic that increases cohesion is *Split Module* where a large non-cohesive module is split into smaller, more cohesive ones. This is a typical activity undertaken in refactoring but it can also be done proactively in design. Several modifiability tactics, as shown in [Figure 3.7](#), directly address reducing coupling, such as: *Encapsulate*, *Use an Intermediary*, *Abstract Common Services*, and *Restrict Dependencies*. So, these tactics, and their related patterns, should be highly prioritized when choosing architectural elements. For example, the *Client-Server* or *Publish-Subscribe* or *Microservices* patterns all serve to reduce binding among architectural elements.

Also, when choosing and instantiating elements, a wise architect will devote enough time to study the pros and cons of the alternatives that are considered before decisions are made. This may involve, for example, devoting time to studying a particular technology, building prototypes that allow a hypothesis to be tested, or doing lightweight architecture analysis, so that a decision can be made based on well founded arguments.

For the scenario described in step 2, the decision was to architect the system around the event-based CQRS pattern (see [8.3.2](#)).

Step 6:Sketch views

When we sketch views, we normally include the rationale for the design decisions taken. This rationale can help others in assessing why a design choice was made—perhaps it was an arbitrary choice, or one that was rushed. In such a case it might be revisited, particularly if it is causing

problems and incurring debt. Furthermore, the rationale might include assumptions about the evolutionary path of the system, or about non-obvious dependencies. By consciously adding information such as this, debt can be minimized or appropriately dealt with in the future. This rationale will also help guide and constrain implementation choices.

Step 7:Perform analysis

If this is greenfield development the analysis undertaken can be scenario-based, where modifiability scenarios are mapped on to the architectural description to determine how well they will be satisfied. Scenario-based analysis is relatively low-cost and easy to perform (see [11.6](#)). It requires no tooling more sophisticated than a whiteboard. If, on the other hand, this is brownfield development, in addition to the above, the team can mine the project's existing data. For example, project members can mine the revision history to pinpoint the areas of the architecture that are incurring the greatest effort—that is, the greatest numbers of changes and churn. They can also run analysis tools over the existing codebase to identify architectural flaws, such as cyclic dependencies, modularity violations, unstable interfaces and so forth. These identified problem areas can then be made the focus of further design (and perhaps refactoring) activities.

Considering the scenario described in step 2, the decision of architecting the system around the event-based CQRS pattern is deemed appropriate to support QA-10. The price change events can be fed to an analytics tool from the cloud provider, or an additional microservice focused on this task can be created. The introduction of either of these two

solutions does not impact the existing components in the system and, furthermore, it does not require the system to be shut down as the analytics component can read historic events from the log. A scenario-based walkthrough describing this approach convinced the stakeholders that this was a good solution.

10.5 Summary

In this chapter we have given a brief overview of technical debt and zoomed in on the kind of debt that we are most concerned about: design debt. This debt is ubiquitous in software systems and can, over time, cause a code base to collapse under its own weight. We argued that this “entropy” is caused, often unintentionally, by bad design choices, and by developers’ actions that undermine the coupling and cohesion of the software. We further argued that this can only be counteracted by an investment in refactoring. But it is often difficult to get management approval for this refactoring because the costs of technical debt are usually invisible, and most projects do not have the discipline, and do not collect the data, to make the business case for paying down the debt.

Finally, we showed how ADD can be used to proactively design with technical debt avoidance or mitigation in mind. All it takes is the right mindset. The steps of ADD can be tailored with little effort, to focus on avoiding and remediating debt. That is the good news.

10.6 Further Reading

A brief introduction to the metaphor of technical debt and its consequences can be found in:

Philippe Kruchten, Robert L Nord, Ipek Ozkaya, "Technical debt: From metaphor to theory and practice", *IEEE Software*, 29:6, 2012, 18-21.

Comprehensive overviews of technical debt in its many dimensions and manifestations can be found in: N. Ernst, J. Delange, R. Kazman, *Technical Debt in Practice—How to Find It and Fix It*, MIT Press, 2021 and in P. Kruchten, R. Nord, I. Ozkaya, *Managing Technical Debt: Reducing Friction in Software Development*, Addison-Wesley, 2019.

A discussion of the survey of practitioners on technical debt carried out by the Software Engineering Institute can be found at: <https://insights.sei.cmu.edu/blog/a-field-study-of-technical-debt/>

The tactics and patterns mentioned in this chapter, and many more as well, can be found in the 4th edition of *Software Architecture in Practice*: L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, 4th ed., Addison-Wesley, 2021

Manny Lehman's laws of software evolution are described in: M. Lehman, "On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle", *Journal of Systems and Software*. 1: 213-221, 1980.

A case study of an organization successfully refactoring to pay down technical debt can be found in: M. Nayebi, Y. Cai, R. Kazman, G. Ruhe, Q. Feng, C. Carlson, F. Chew, "A Longitudinal Study of Identifying and Paying Down Architectural Debt", *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2019.

And a discussion of architectural flaws, their costs, and their detection can be found in: L. Xiao, R. Kazman, Y. Cai, R. Mo, Q. Feng, "Detecting the Locations and Predicting the Costs of Compound Architectural Debts", *IEEE Transactions on Software Engineering*, 48:9, Sept., 2022,

and in R. Mo, Y. Cai, R. Kazman, L. Xiao, Q. Feng
“Architecture Anti-patterns: Automatically Detectable
Violations of Design Principles”, *IEEE Transactions on
Software Engineering*, 47:5, May, 2021.

The DV8 design analysis tool was described in Y. Cai, R. Kazman, “DV8: Automated Architecture Analysis Tool Suites”, *Proceedings of TechDebt 2019 International Conference on Technical Debt (Tools Track)*, (Montreal, Canada), May 2019. A trial version is available from archdia.com.

An analysis of technical debt detection tools, and how they report startlingly inconsistent results, can be found in: J. Lefever, Y. Cai, H. Cervantes, R. Kazman, H. Fang, “On the Lack of Consensus Among Technical Debt Detection Tools”, *Proceedings of the 43rd International Conference on Software Engineering*, (ICSE) 2021.

10.7 Discussion Questions

1. Technical debt can come in many forms—requirements debt, code debt, testing debt, design debt, deployment debt, and documentation debt—which can affect many aspects of a project. Which of these do you think is the most important and most problematic for a software project? Why?
2. When is it a good time to incur design debt and when is it a bad time?
3. Should you document your design debt? If so, what would this documentation look like? And when should you create it?
4. What kinds of stakeholders should you engage so that you can get the best scenarios for your system, in terms

of scenarios that will have an impact on your future design debt?

5. When should you analyze your design debt? Every release? Every commit? Some other frequency?

11. Analysis in the Design Process

Design is the process of making decisions; analysis is the process of understanding those decisions, so that the design may be evaluated. To reflect on this intimate relationship, we now turn our attention to questions of why, when, and how to analyze architectural decisions *during* the design process. We contend that it is impossible to do design without doing some analysis. But frequently this analysis is ad hoc, or purely intuitive. In many cases that is good enough, but not always. It makes sense to be clear about if, when, and where we need to put a bit more thought and a bit more analysis into the decisions we make, *as we are making them*. In this chapter we will take a quick look at various techniques for analysis, discuss when they can be enacted, and their costs and benefits.

Why read this chapter?

While this is a book focused on architectural design, we have always believed that design and analysis are two sides of the same coin. Besides performing design, another essential task for architects is to perform analysis. We hope to convince you in this chapter that this is not an onerous

duty or one that will greatly increase the time and effort devoted to design. The benefits of analysis greatly outweigh the costs, in terms of early problem resolution and greater confidence in the outputs of the design process.

11.1 Analysis and design

Analysis is the process of breaking a complex entity into its constituent parts as a means of understanding it. The opposite of analysis is synthesis. Analysis and design are therefore intertwined activities. During the design process, the activity of analysis can refer to several aspects:

- *Studying the inputs to the design process to understand the problem whose solution you are about to design.* This includes giving priorities to the drivers as discussed in [section 4.2.2](#). This type of analysis is performed in steps 1 and 2 of ADD.
- *Studying the alternative design concepts that you identified to solve a design problem to select the most appropriate one.* In this situation, analysis leads you to provide concrete evidence for your choices. This is performed in step 4 of ADD and was discussed in [section 4.2.4](#).
- *Ensuring that the key decisions that have been made during an iteration of the design process are appropriate ones.* This is the type of analysis that you perform in step 7 of ADD.

The decisions that you make when designing the architecture are not only critical to achieve the quality attribute responses but, frequently, the cost associated with correcting them at a later time can be significant, since they may affect many parts of the system. For these reasons, it is necessary to perform analysis *during* the design process, so

that problems can be identified, possibly quantified, and corrected quickly. Remember, being too confident and following your gut feelings may not be the best idea (see the sidebar: “I believe” isn’t good enough). Fortunately, if you have followed the recommendations that we have given up to this point, you should be able to conduct analysis—either by yourself or with the help of peers—by using the sketches and rationale that you have been capturing as you make design decisions.

“I believe” isn’t good enough

Even if you are following a systematic approach to designing your architecture and you are using design concepts from well established sources, and even if you have nice looking diagrams where you have represented your structures, nothing really guarantees that the decisions you are making will actually satisfy a particular architectural driver. Certain quality attributes are critical to the success of your system and particularly for these, the justification of “I believe” is not good enough.

Studies of practicing software architects have shown that most follow an “adequacy” approach to making design decisions—that is, they adopt the first decision that appears to meet their needs. Architects, like all people, are subject to confirmation bias and availability bias. And they often have no strong rationale to substantiate those decisions other than their gut feelings, their beliefs. This means that important decisions are frequently made after insufficient reasoning, and this can add risk to a system.

For drivers that are truly critical to your system, you owe it to yourself and to your organization to perform a more detailed analysis rather than just trusting your gut feeling, relying on analogy and history, or performing a couple of

superficial tests to show that your drivers are satisfied. There are several options that you can choose from to deepen your analysis and hence support your rationale for the decisions you've made:

- *Analytic models*: These are well established mathematical models that allow quality attributes such as performance or availability to be studied. They include Markov and statistical models for availability, and queuing and real-time scheduling theory for performance. These models are highly mature but may require considerable education and training to be used adequately.
- *Checklists*: Checklists are useful tools that allow you to ensure, in a systematic way, that important decisions and considerations are not forgotten. There are checklists for particular quality attributes in the public domain, such as the OWASP checklist which guides in performing black box security testing of web applications. Also, your organization may develop proprietary checklists that are specific to the application domains that you are developing. Shortly we will present tactics-based questionnaires, which are a variety of checklist for the most critical system quality attributes, based on a knowledge of tactics.
- *Thought experiments, Reflective Questions, Back-of-the-Envelope Analyses*: Thought experiments are informal analyses performed by a small group of designers where important scenarios are studied to identify potential problems. For example, you may use a sequence diagram produced inside step 5 of ADD and perform a walk-through of the interaction of the objects that support the scenario modeled in the diagram with a colleague. Reflective questions (which we will discuss in section 11.5) are questions that challenge the

assumptions on which a decision is made. Back-of-the-envelope analyses are rough calculations that are less precise than analytic models, but which can be performed quickly. These calculations, frequently based on analogy with other similar systems, or based on prior experience, are useful to obtain ballpark estimates for desired quality attribute responses. For example, by summing the latencies of a number of processes in a pipeline, you can derive a first-order estimate of the end-to-end latency.

- *Prototypes, simulations, and experiments:* Purely conceptual techniques for analyzing a design are sometimes inadequate to accurately understand if certain design decisions are appropriate or not, or whether you should favor one particular technology over another. In situations like these, the creation of prototypes, simulations or experiments can be an invaluable option to obtain a better understanding. For example, in the back-of-the-envelope estimate of latency described above, you may not have taken into account that several of the processes are sharing (and hence competing for) the same resources and so you can not simply sum their individual latencies and expect to get accurate results. The creation of prototypes or simulations provide a deeper understanding of system dynamics, but they may require a significant effort that needs to be considered in the project plan.

As always, there is no one technique that is consistently better than the others. Thought experiments and back-of-the-envelope calculations are inexpensive and can be done early in the design process, but their validity may be questionable. Prototypes, simulations, and experiments typically produce much higher fidelity results, but at a far greater cost and with a bigger impact on your schedule. The

choice of which to do depends on the context, the risk involved, and the priorities of your quality attributes.

However, you should keep in mind that applying *any* of these techniques will be helpful in going from “I believe” (that my design is appropriate) to an approach that is backed by documented evidence and argumentation.

11.2 Why Analyze?

As we said above, analysis and design are two sides of the same coin. Design is (the process of) making decisions. Analysis is (the process of) understanding the consequences of those decisions in terms of cost, schedule, and quality. No sensible architect would make any decision, or at least any non-trivial decision, without first attempting to understand the implications of that decision: its near-term and possibly long-term consequences. Architects may make thousands of decisions in the course of designing a large project and clearly not all of them matter. Furthermore, not all of the decisions that matter are carriers of quality attributes. Some may be about which vendor to select, or what coding convention to follow, or which programmer to hire or fire, or which IDE to use—important decisions, to be sure, but not directly linked to a quality attribute outcome.

But, just as clearly, some of these decisions *will* affect the achievement of quality attributes. When the architect breaks down the development into a system of layers or modules, or both, this decision will affect how a change will ripple through the code base, who needs to talk to who when modifying a feature or fixing a bug, how easy or difficult it is to distribute or outsource some of the development, how easy it is to port the software to a different platform, and so forth. When the architect chooses a distributed resource management system, how it

determines which services are leaders and which are followers, how it detects failures, and how it detects resource starvation, will affect the availability of the system.

When and why do we analyze during the design process? First, we analyze *because we can*. An architecture specification, whether it is just a whiteboard sketch or something that has been more formally documented and circulated, is the first artifact supporting an analysis that sheds insight into quality attributes. Yes, we can analyze requirements, but we mainly analyze them for consistency and completeness. Until we translate those requirements into structures resulting from design decisions, we have little to say about the actual consequences of those decisions, their costs and benefits, and the tradeoffs among them.

Second, and more to the point, we analyze because it is a prudent way of informing decisions and *managing risk*. No design is completely without risk, but we want to ensure that the risks that we are taking on are commensurate with our stakeholders' expectations and tolerances. For a banking application or a military application we would expect our stakeholders to demand low levels of risk, and they should be willing to pay accordingly for higher levels of assurance. For a startup company, where time to market is of the essence and budgets are tight, we might be prepared to accept far higher levels of risk. As with every important decision in software engineering: it depends.

Finally, analysis is the key to evaluation. Evaluation is the process of determining the value of something. Companies are evaluated to determine their share price. A company's employees are evaluated annually to determine their raises. In each case the evaluation is built upon an analysis of the properties of the company or employee.

11.3 Analysis Techniques

Different projects will deserve (and possibly demand) different responses to risk. Fortunately we, as architects, have a wide variety of practices and methods at our disposal to analyze architectures. With a bit of planning we can match our risk tolerance with a set of analysis techniques that both meet our budget and schedule constraints and provide reasonable levels of assurance. The point here is that analysis does not have to be costly or complex. Just asking thoughtful questions is a form of analysis, and that is pretty inexpensive. Building a simple prototype is more expensive, but in the context of a large project this may be an analysis technique that is worth the additional expense, in terms of how it explores and mitigates risks.

Examples of (relatively economical, relatively low ceremony) analysis already in widespread use include design reviews and scenario-based analyses, code reviews, pair programming, and Scrum Retrospective meetings. Other commonly used analysis techniques, although somewhat more costly, include prototypes (throw-away or evolutionary) and simulations.

At the high end of expense and complexity we can build formal models of our systems and analyze them for properties such as latency or security or safety. And, finally, when there is a candidate implementation or a fielded system, we can perform experiments, including instrumenting running systems and collecting data, ideally from executions of the system that reflect realistic usages.

As indicated in [Table 11.1](#), the cost of these techniques typically increases as you proceed through the software development lifecycle. A prototype or experiment is more expensive than a checklist which is more expensive than

experience-based analogy. However, this expected cost is strongly correlated with the confidence that you have in the analysis results. Unfortunately there is no free lunch!

Table 11.1 *Analysis at Different Stages of the Software Lifecycle*

Life-cycle Stage	Form of Analysis	Cost	Confidence
Requirements	Experience-based Analogy	Low	Low–High
Requirements	Back-of-the-envelope Analysis	Low	Low–Medium
Architecture	Thought Experiment / Reflective Questions	Low	Low–Medium
Architecture	Checklist-based Analysis	Low	Medium
Architecture	Tactics-based Analysis	Low	Medium
Architecture	Scenario-based Analysis	Low	Medium
Architecture	Analytic Model	Low–Medium	Medium
Architecture	Simulation	Medium	Medium
Architecture	Prototype	Medium	Medium–High
Implementation	Experiment	Medium–High	Medium–High

Fielded System	Instrumentation	Medium-High	High
----------------	-----------------	-------------	------

11.4 Tactics-based Analysis

Architectural tactics (discussed in [Chapter 3](#)) have been presented thus far as design primitives. However, since these taxonomies are intended to cover the entire space of architectural design possibilities for managing a quality attribute, we can use them in an analysis setting as well. Specifically, we can use them as guides for interviews or questionnaires. These interviews help you to, as an analyst, to gain rapid insight into the architectural approaches taken, or not taken.

Consider, for example, the tactics for availability, shown in [Figure 3.7](#). Each of these tactics is a design option for the architect who wants to design a highly available system. But, used in hindsight, they represent a taxonomy of the entire design space for availability and hence can be a way of gaining insight into the decisions made, and not made, by the architect. To do this we simply turn each tactic into an interview question. For example, consider the (partial) set of tactics-inspired availability questions in [Table 11.2](#).

Table 11.2 *Example Tactics-based Availability Questions*

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Detect Faults	Does the system use ping/echo to detect a failure of a component or connection, or network congestion?				
	Does the system use a component to monitor the state of health of other parts of the system. A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack.				
	Does the system use a heartbeat —a periodic message exchange between a system monitor and a process—to detect a failure of a component or connection, or network congestion?				

	Does the system use a timestamp to detect incorrect sequences of events in distributed systems?				
	Does the system use voting to check that replicated components are producing the same results. The replicated components may be: identical replicas, functionally redundant, or analytically redundant.				
	Do you use exception detection to detect a system condition that alters the normal flow of execution, e.g. system exception, parameter fence, parameter typing, timeout?				
	Can the system do a self test to test itself for correct operation?				
Recover	Does the system employ				

from Faults (Preparation and Repair)	<p>a redundant spare? That is, does it use a configuration in which one or more duplicate components can step in and take over the work if the primary component fails?</p> <p>Does the system employ exception handling to deal with faults?</p> <p>Typically the handling involves either reporting the fault or handling it, potentially masking the fault by correcting the cause of the exception and retrying.</p>			
	<p>Does the system employ rollback, so that it can revert to a previously saved good state (the “rollback line”) in the event of a fault?</p>			

When used in an interview setting, we can record whether or not each tactic is supported by the system's architecture, according to the opinions of the architect. If we are analyzing an existing system we can additionally investigate:

- whether there are any obvious risks in the use (or non-use) of this tactic. If the tactic has been used we can record here how it is realized in the system (e.g. via custom code, generic frameworks, externally produced components, etc.). For example, we might note that the Redundant Spare tactic has been employed by replicating the query microservice in the case study in [chapter 8](#).
- the specific design decisions made to realize the tactic and where in the code-base the implementation (realization) may be found. This is useful for auditing and architecture reconstruction purposes. Continuing the example from the previous bullet, we might probe how many replicas of the microservice have been created, and where these replicas are located (i.e. in different regions or availability zones, etc.).
- any rationale or assumptions made in the realization of this tactic. For example, we might assume that there will be no common-mode failure, and so it is acceptable that the replicas are identical microservices, running on identical execution environments.

While this interview-based approach might sound simplistic, it can actually be quite powerful and insightful. In your daily activities as an architect you likely do not take the time to step back and consider the bigger picture. A set of interview questions such as those shown in [Table 11.2](#) force you to do just that. And this is quite efficient: a typical interview for a single quality attribute takes between 30 and 90 minutes.

A set of tactics-based questionnaires, covering the ten most important system quality attributes—availability, energy efficiency, deployability, integrability, modifiability, performance, safety, security, testability, and usability—can be found in [Appendix A](#). In addition, we have included an eighth questionnaire, on DevOps, as an example of how you

can combine the other (more fundamental) questionnaires to create a new questionnaire to address a new set of quality concerns.

11.5 Reflective Questions

Similar to the tactics-based interviews, a number of researchers have advocated the practice of asking (and answering) reflective questions to augment the design process. The idea behind this process is that we actually think differently when we are problem-solving and when we are reflecting. For this reason, these researchers have advocated a separate “reflection” activity in design that challenges the decisions made, and that challenges us to examine our biases.

Architects, like all humans, are subject to bias. For example, we are subject to confirmation bias—the tendency to interpret new information in a way that confirms our preconceptions—and we are subject to anchoring bias—the tendency to rely too heavily on the first piece of information that we receive when investigating a problem and using this information to filter and judge any subsequent information. Reflective questions help to uncover and counter such biases in a systematic way, and this can lead us to revise our assumptions and hence our designs.

One can and should reflect on a system’s context and requirements (are the contexts and requirements identified relevant, complete, and accurate), design problems (have they been properly and fully articulated), design solutions (are they appropriate given the requirements), and design decisions (are they principled and justified). Examples of reflective questions that they propose include:

- What assumptions are made? Do the assumptions affect the design problem? Do the assumptions affect

the solution option? Is an assumption acceptable in a decision?

- What are the risks that certain events would happen? How do the risks cause design problems? How do the risks affect the viability of a solution? Is the risk of a decision acceptable? What can be done to mitigate the risks?
- What are the constraints imposed by the contexts? How do the constraints cause design problems? How do the constraints limit the solution options? Can any constraints be relaxed when making a decision?
- What are the contexts and the requirements of this system? What does this context mean? What are the design problems? Which are the important problems that need to be solved? What does this problem mean? What potential solutions can solve this problem? Are there other problems to follow up in this decision?
- What contexts can be compromised? Can a problem be framed differently? What are the solution options? Can a solution option be compromised? Are the pros and cons of each solution treated fairly? What is an optimal solution after considering tradeoffs?

Of course, you might not employ all of these questions, and you would not employ this technique for every decision that you make. But, used judiciously, these kinds of questions are fast and economical, and can help you to reflect mindfully on the decisions that you are making.

11.6 Scenario-Based Design Reviews

Comprehensive scenario-based design reviews, such as the ATAM, have typically been conducted outside the design

process. An ATAM is an example of a comprehensive architecture evaluation (see the sidebar: The ATAM).

An ATAM review, as it was initially conceived, was a “milestone” review. When an architect or other key stakeholder believed that there was enough of an architecture or architecture description to analyze, an ATAM meeting could be convened. This might occur when an architectural design had been done but before much, if any, implementation had been completed. Or, more commonly, it occurred when an existing system was in place and some stakeholders wanted an objective evaluation of the risks of the architecture, before committing to it, evolving it, acquiring it, and so forth.

Sidebar: The ATAM

The ATAM—Architecture Tradeoff Analysis Method—is an established method for analyzing architectures, driven by scenarios. The purpose of the ATAM is to assess the consequences of architectural decisions in light of quality attribute requirements and business goals.

The ATAM brings together three groups in an evaluation:

- a trained evaluation team
- an architecture’s “decision makers”
- representatives of the architecture’s stakeholders

The ATAM helps stakeholders ask the right questions to discover potentially problematic architectural decisions—risks. These discovered risks can then be made the focus of mitigation activities such as further design, further analysis, prototyping, and implementation. In addition, design tradeoffs are often identified—hence the name of the method. The purpose of the ATAM is *not* to provide precise

analyses: the method typically takes place over two two-day meetings and this (relatively) short timeframe does not permit a deep dive into any specific concern. Those kinds of analyses are, however, appropriate as part of the risk mitigation activities that could follow and be guided by an ATAM.

The ATAM can be used throughout the software development life cycle. It can be used:

- after an architecture has been specified but there is little or no code
- to evaluate potential architectural alternatives
- to evaluate the architecture of an existing system

The outputs of the ATAM are as follows:

- A concise presentation of the architecture. The architecture is presented in one hour.
- A concise articulation of the business goals for the system under scrutiny. Frequently, the business goals presented in the ATAM are being seen by some of the assembled participants for the first time and these are captured in the outputs.
- A set of prioritized quality attribute requirements, expressed as scenarios.
- A mapping of architectural decisions to quality requirements. For each quality attribute scenario examined, the architectural decisions that help to achieve it are identified and recorded.
- A set of sensitivity and tradeoff points. These are architectural decisions that have a marked effect on one or more quality attributes.

- A set of risks and non-risks. A risk is defined as an architectural decision that may lead to undesirable consequences in light of quality attribute requirements. A non-risk is an architectural decision that, upon analysis, is deemed safe. The identified risks form the basis for an architectural risk mitigation plan.
- A set of risk themes. The evaluation team examines the full set of discovered risks to identify overarching themes that reveal systemic weaknesses in the architecture (or even in the architecture process and team). If left untreated these will threaten the project's business goals.

There are also intangible results of an ATAM-based evaluation. These include: a sense of community on the part of the stakeholders, open communication channels between the architect and the stakeholders, a better overall understanding of the architecture and its strengths and weaknesses. While these results are difficult to measure, they are no less important than the others and often are the longest-lasting.

An ATAM takes place in 4 phases. The first phase (phase 0) and the final phase (phase 3) are managerial: setting up the evaluation at the start and reporting results and follow-on activities at the end. The middle phases (phase 1 and phase 2) are when the actual analysis takes place. The steps of the ATAM enacted in phases 1 and 2 are as follows:

1. Present the ATAM
2. Present business drivers
3. Present architecture
4. Identify architectural approaches
5. Generate quality attribute utility tree

6. Analyze architectural approaches
7. Brainstorm and prioritize scenarios
8. Analyze architectural approaches
9. Present results

In phase 1 we enact steps 1-6 with a small, internal group of stakeholders, typically just the architect, project manager and perhaps one or two senior developers. In phase 2 we invite a larger group of stakeholders to attend—all the people who attended phase 1 plus external stakeholders, such as customer representatives, end-user representatives, quality assurance, operations, and so forth. In phase 2 we review steps 1-6 and enact steps 7-9.

The actual analysis takes place in Step 6, where we analyze architectural approaches by asking the architect to map the highest priority scenarios, one at a time, onto the architectural approaches that have been described. During this Step the analysts ask probing questions, motivated by a knowledge of quality attributes, and risks are discovered and documented.

The idea of having a distinct evaluation activity once the architecture is “done”, is a poor fit with the way that most organizations operate today. Today, most software organizations are practicing some form of agile or iterative development. There is no distinct monolithic “architecture phase” in agile processes. Rather, architecture and development are co-created in a series of sprints. For example, as we will discuss in [section 12.1](#), many agile thought leaders are promoting practices such as “disciplined agility at scale”, the “walking skeleton”, and the “scaled agile framework”, all of which embrace the idea that architectures continuously evolve in relatively small increments, addressing the most critical risks. This may be

aided by developing a small proof-of-concept or MVP (minimum viable product), or doing strategic prototyping.

To better align with this view of software development, a lightweight scenario-based peer review method, based on the ATAM, has been created. The lightweight ATAM follows all the steps of the full ATAM but limits their scope thus shortening the time allotted to each step. This modified method consists of just a single analysis phase, unlike the full ATAM which has two phases with two groups of stakeholders, one internal and one external. As a consequence, a lightweight ATAM can be conducted in a half-day meeting. And you can do this internally, using just project members. Of course an external review gives more objectivity, and may produce better results, but this may be too costly or infeasible due to schedule or IP constraints. A lightweight ATAM therefore provides a reasonable middle ground between the costly but objective and comprehensive ATAM and doing no analysis whatsoever, or only doing *ad hoc* analysis.

[Table 11.3](#) outlines an example schedule for a lightweight ATAM conducted by project members on their own project.

Table 11.3 *A Typical Agenda for a Lightweight ATAM*

Step	Time Allotted	Notes
1: Present business drivers	0.25 hrs	The participants are expected to understand the system and its business goals and their priorities. Fifteen minutes is allocated for a brief review to ensure that these are fresh in everyone's mind and that there are no surprises.
2: Present Architecture	0.5 hrs	Again, all participants are expected to be familiar with the system and so a brief overview of the architecture is presented and 1–2 scenarios are traced through the documented architecture views.
3: Identify Architectural Approaches	0.25 hrs	The architecture approaches for specific quality attribute concerns are identified by the architect. This may be done as a portion of Step 2.
4: Generate Quality Attribute Utility Tree	0.5 hrs	Scenarios might already exist: if so, use them. A utility tree might already exist; if so, the team reviews this and updates it, if needed.
5: Analyze Architectural Approaches	2.0 hrs	This step—mapping the highly ranked scenarios onto the architecture—consumes the bulk of the time and can be expanded or contracted as needed.

6: Present Results	0.5 hrs	At the end of the evaluation, the team reviews the existing and newly discovered risks and tradeoffs and discusses priorities.
TOTAL	4 hrs	

A half-day review such as this is similar, in terms of effort, with other quality assurance efforts that are typically conducted in a development project, such as code reviews, inspections, and walkthroughs. For this reason, it is easy to schedule this in a sprint, particularly in those sprints where architectural decisions are being made, challenged, or changed.

11.7 Summary

No responsible programmer would field code that they had not tested. But architects and programmers regularly commit to (implement) architectural decisions that have not been analyzed. Why the dichotomy? Surely if testing code is important then “testing” the design decisions you have made is an order of magnitude more important, as these decisions often have long-term, system-wide, and significant impacts.

The most important message of this chapter is that design and analysis are not separate activities. Every important design decision that you make should be analyzed and there are techniques for doing this continuously, in a relatively disruption-free manner, as part of the process of designing and evolving a system.

The interesting questions are not whether to analyze, but rather how much to analyze and when. Analysis is inherent in doing good design, and it should be a continuous process.

11.8 Further Reading

The sets of architectural tactics used here have been documented in L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice* (4th edition), Addison-Wesley, 2021. The availability tactics were first created in J. Scott, R. Kazman, “Realizing and Refining Architectural Tactics: Availability”, CMU/SEI-2009-TR-006, 2009.

The idea of reflective questions was first introduced in M. Razavian, A. Tang, R. Capilla, and P. Lago, “In Two Minds: How Reflections Influence Software Architecture Design Thinking,” VU University Amsterdam, Tech. Rep. 2015-001, April 2015. And the idea that software designers satisfice—that is, they look for a “good enough”, as opposed to an optimal solution—has been discussed in A. Tang, H. van Vliet. “Software Designers Satisfice”. *European Conference on Software Architecture (ECSA 2015)*, 2015.

The ATAM was comprehensively described in P. Clements, R. Kazman, M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2001 and the lightweight ATAM is explained in L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice* (4th edition), Addison-Wesley, 2021. In addition ATAM-style peer reviews have been described in: F. Bachmann, “Give the Stakeholders What They Want: Design Peer Reviews the ATAM Style”, *Crosstalk*, November/December, 2011.

A discussion of how to use analysis models for reasoning about the quality attribute of Robustness can be found in

this technical report: R. Kazman, P. Bianco, S. Echeverria, J. Ivers, "Robustness", CMU/SEI-2022-TR-004, 2022.

Reviews have social as well as technical aspects. A discussion of some of the non-technical (social, psychological, managerial) aspects of reviews can be found in: R. Kazman and L. Bass, "Making architecture reviews work in the real world," *IEEE Software*, vol. 19, no. 1, pp. 67-73, Jan.-Feb. 2002

11.9 Discussion questions

1. What is the difference between requirements analysis and analysis during the design process?
2. What are the benefits of using a checklist-based approach, such as tactics-based analysis versus a more open analysis approach such as scenario-based design reviews?
3. Consider the case study of [chapter 9](#). Propose some reflective questions that might be relevant to it. These reflective questions could be associated with the requirements or with the design decisions.
4. Consider the case study of [chapter 8](#) and select a quality attribute category from the ones listed in [section 8.2.2](#). Use the tactics-based questionnaire associated with this category and identify the tactics that are supported or not supported. Also fill in the remaining columns of the questionnaire.
5. Consider the case study of [chapter 8](#). Choose the most important scenarios and perform a scenario-based analysis of the design. Can you identify at least three risks in the decisions that were made by the architect? Justify why they are risks.

12. The Architecture Design Process in the Organization

In the previous chapters of this book, we focused our attention on designing architectures, and the concerns surrounding architectural design. In this chapter we focus our attention on how architectural design integrates both within the development lifecycle and the organization as a whole.

Why read this chapter?

An architect's job is not purely technical. An architect is the fulcrum between business requirements on one hand, and technical solutions on the other hand. Architectures exist within an organization. The effective architect is aware of this context—such as life-cycle concerns and existing team structures—and creates architectures that are well-aligned with their contexts. Having read this chapter, you should be aware of the many forces and concerns that an architecture influences and is influenced by, and you should be better able to be a positive force for good in your organization.

12.1 Architecture Design and the Development Life Cycle

As we saw in [section 2.4.1](#), architecture design can serve different purposes. Here we are interested in four contexts where designs and design decisions are typically made. The first context is before the development of the software system begins, where an architectural design is created to support estimation of project cost, effort, and schedule. The second and third contexts are when an architectural design is being created to support agile or sequential development. And the final context is when architectural decisions need to be made to support a project's testing strategy.

12.1.1 Designing to support estimation

In many types of development projects, organizations typically need to provide an initial estimate of the cost, effort, and schedule of the project. In the context of product-oriented development, the word project used here refers to the delivery of a particular milestone of the product. This estimation phase frequently determines whether or not the project will actually take place, as shown in [Figure 12.1](#). The estimation phase can be done both for internal and external projects.

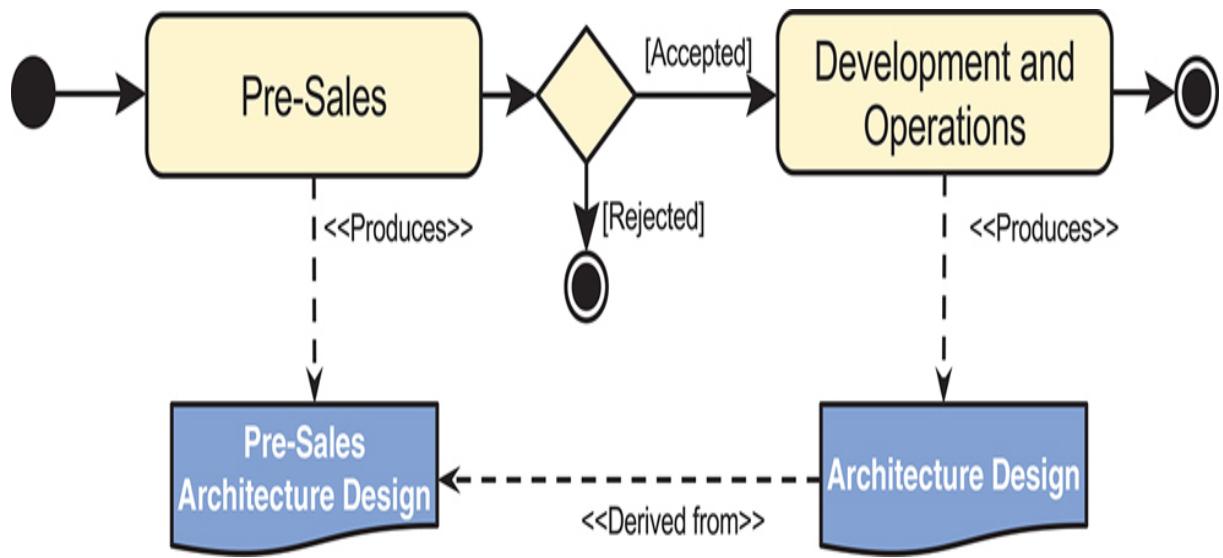


FIGURE 12.1 The relationship between estimation and project development

Frequently, estimation activities must be performed in a short time period, and the information that is available to inform this process is always limited. For example, typically only high-level requirements or features (rather than detailed user stories) are available at this phase. Also, quality attributes are frequently not detailed with precise measurements; only a rough idea of the important quality attributes is available.

The problem with limited information is that the estimate that is produced frequently has a high level of variance, as illustrated by the *cone of uncertainty* depicted in [Figure 12.2](#). This cone depicts the uncertainty surrounding estimates in a project, typically those of cost and schedule, but also risk. All of these estimates get better as a project progresses, and the cone narrows. When the project is done, uncertainty is zero. The issue for any development methodology is how to narrow the cone of uncertainty earlier in the project's life cycle.

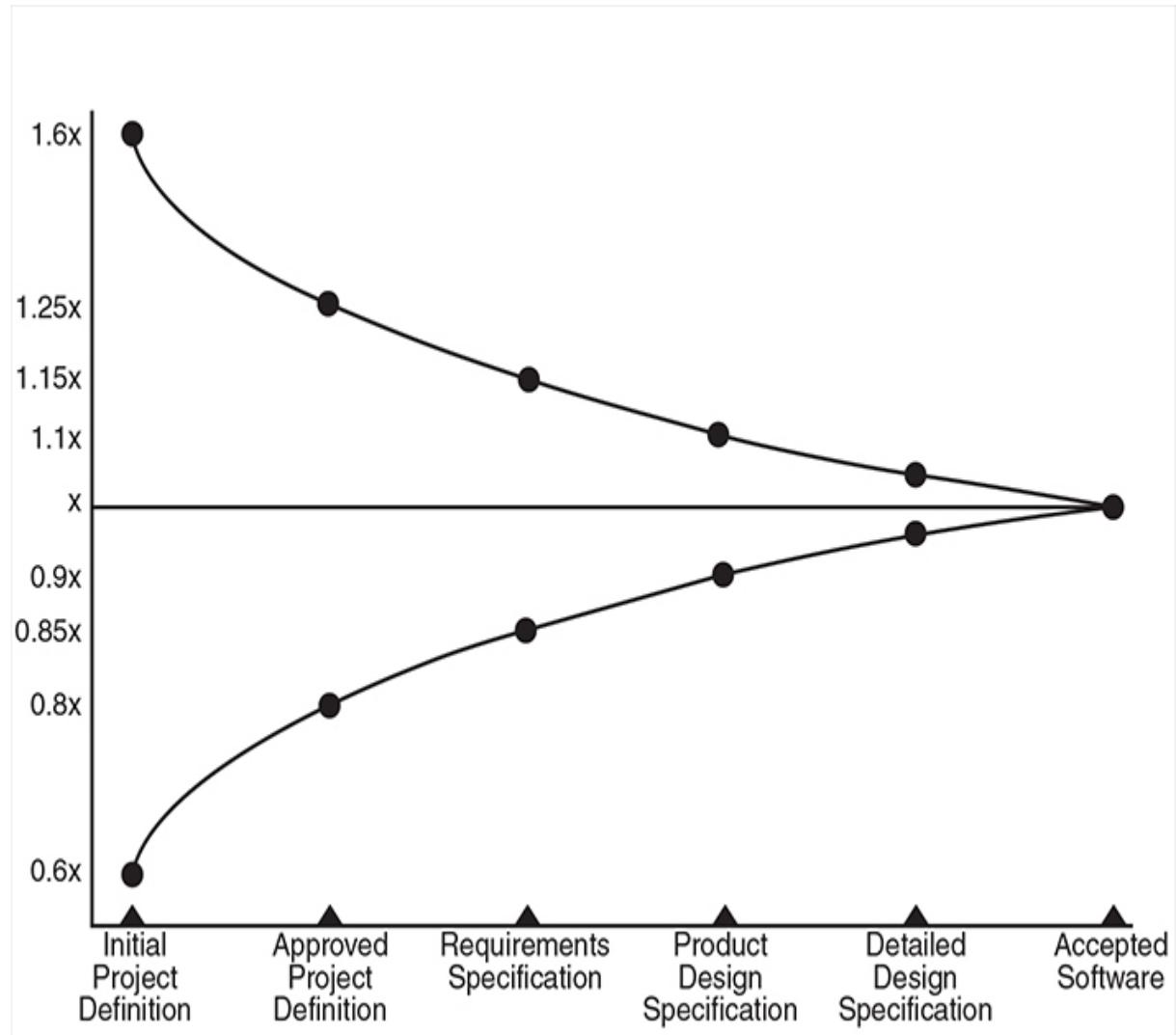


FIGURE 12.2 Example cone of uncertainty

Architectural practices can be applied in the estimation phase to gain understanding and help reduce the variance in the estimates:

- Architectural drivers can be identified in this phase. Even if it may be complicated to describe detailed quality attribute scenarios at this point, the most important quality attributes with initial measures and constraints should be identified. Following the principles from QAW (see 2.4.2), quality attributes that are important to different stakeholders should be identified.

- ADD can be used to produce an initial architecture that is then used as the basis for early cost and schedule estimates.
- Sketches of this initial architecture are useful for communication with the customer. They are also useful as a basis to perform lightweight evaluations of this initial design.

Generating an initial architecture allows estimation to be performed using the “standard components” technique. Standard components are a type of proxy; they include front ends, components associated with specific layers, services, or microservices, among other things. When estimating based on standard components, companies typically build databases that contain historical measurements of size, effort, and data volumes for components that have been built into previously developed systems. To estimate with standard components, you need to identify the components that will be required for the problem that you are trying to solve, and then use historical data (or some other technique such as Wideband Delphi) to estimate the size of these components. The total size can then be translated into effort, and these estimates can be rolled up to produce a project-level time and cost estimate.

Identifying the components that are required to create estimates with this technique can be achieved in a short time frame through the use of ADD. This approach is similar to what we recommended for the design of greenfield systems in [section 4.3.1](#):

- The goal of your first design iteration should be to address the concern of establishing an initial overall structure for the application. The reference architecture, if you employ one, dictates the types of standard components that will be used in the estimation. At this

point, the most relevant technologies to use in the project can also be selected, particularly if your historical data is tied to specific technologies.

- The goal of your second design iteration should be to identify components to support all of the functionality that needs to be considered for the estimation. As opposed to what we discussed for the design of greenfield systems, when designing to produce an estimate, you need to consider more than just primary functionality. To identify the standard components, you need to consider *all* of the important functional requirements that are part of the scope and map them to the structure that you defined in the first iteration. Doing so ensures that you identify a majority of components needed to estimate, thus producing a more accurate estimation.

This technique will help you estimate costs and schedule for meeting the most important functional requirements. At this point, however, you will likely not have taken quality attributes into account. As a consequence, you should perform a few more iterations focusing on where you will make design decisions to address the driving quality attributes. If the time available to perform the estimation process is limited, you will not be able to design it in much detail, so the decisions that you should take here are the ones that will have a significant impact on the estimate. In the case of systems that will be deployed in the cloud, at least one iteration should be devoted to designing the cloud architecture of the system, considering aspects such as redundancy, performance and security where possible. Cloud provider calculators, such as the one discussed in [section 7.2.2.1](#), can be used to estimate the costs associated with this initial infrastructure.

When this technique is used in the estimation process, an initial architecture design is produced—the preliminary architecture design (see [Figure 12.1](#)). If the project proposal is accepted by the customer and the project proceeds, this initial architecture can become one of the bases for a contract. This architecture should be used as a starting point in the subsequent architecture design activities that are performed during the Development and Operation phase of the project. In this case, the approach suggested for designing brownfield systems (discussed in [Section 4.3.3](#)) can be used.

Furthermore, the preliminary documentation produced for this initial architecture can be included as part of the technical proposal that is provided to the customer. Finally, this initial architecture design can also be evaluated, preferably before estimation occurs. This can be performed using a technique such as the lightweight ATAM presented in [section 11.6](#).

12.1.2 Designing to support agile development

The relationship between software architecture and agility has been the subject of some debate over the past decade. Although we believe, and much research has shown, that architectural practices and Agile practices are actually well aligned, this position has not always been universally accepted. Here we discuss different aspects which are relevant when designing to support agile development.

12.1.2.1 Approaches to design

Agile practices, according to the original Agile Manifesto emphasize, “Individuals and interactions over processes and tools, working software over comprehensive documentation,

customer collaboration over contract negotiation, and responding to change over following a plan". None of these values is inherently in conflict with architectural practices. So why has the belief arisen—at least in some circles—that the two sets of practices are somehow incompatible? The crux of the matter is the one principle on which Agile practices and architectural practices differ.

The original creators of the Agile Manifesto described 12 principles behind the manifesto. While 11 of these are fully compatible with architectural practices, one of them is not: "The best architectures, requirements, and designs emerge from self-organizing teams." While this principle may have held true for small and perhaps even medium-sized projects, we are unaware of any cases where it has been successful in large projects, particularly those with complex or novel requirements and distributed development. The heart of the problem is this: Software architecture design is "up-front" work. You could always just start a project by coding and doing minimal or no up-front analysis or design. This is what we call the *emergent approach*, as shown in [Figure 12.3b](#). In some cases—small systems, throw-away prototypes, systems where you have little idea of the customer's requirements—this may, in fact, be the optimal decision. At the opposite extreme, you could attempt to collect all the requirements up front, and from that synthesize the ideal architecture, which you would then implement, test, and deploy. This so-called *Big Design Up Front* approach (BDUF; [Figure 12.3a](#)) is usually associated with the classic Waterfall model of software development. The Waterfall model has fallen out of favor over the past decades due to its complexity and rigidity, which led to many well-documented cases of cost overruns, schedule overruns, and customer dissatisfaction. With respect to architectural design, the downside of the BDUF approach is that it can end up producing an extensively documented but untested design

that may not be appropriate. This occurs because problems in the design are often discovered late and may require a lot of rework, or the original design may end up being ignored by developers in their attempts to satisfy system requirements, and the true architecture is thus not documented.

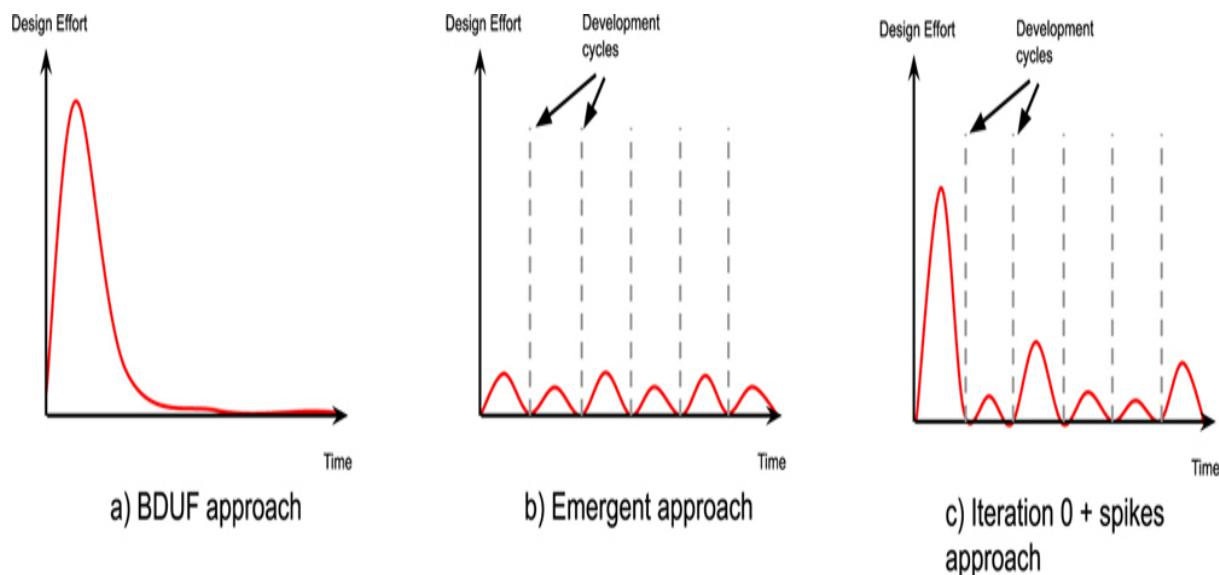


FIGURE 12.3 Three approaches to architectural design

Clearly, neither of these extremes makes sense for most real-world projects, where some (but not all) of the requirements are well understood up front but there is also a risk of doing too much too soon and hence becoming locked into a solution that will inevitably need to be modified, at significant cost. So, the truly interesting question is this: How much up-front work, in terms of requirements analysis, risk mitigation, and architecture, should a project do? Boehm and Turner have presented evidence arguing that there is no single right answer to this question, but that you can find a “sweet spot” for any given project. The “right” amount of up-front work depends on several factors, with the most dominant being project size, but other important factors include requirements

complexity, requirements volatility (related to the novelty of the domain), and degree of distribution of development.

So how do architects achieve the right amount of agility? How do they find the right balance between up-front work and technical debt leading to rework? For small, simple projects, no up-front work on architecture is justifiable. It is easy and relatively inexpensive to turn on a dime and refactor. In more complex or more mature projects, where there is already some understanding of the requirements, begin by performing a few ADD iterations. These iterations can focus on aspects such as:

- Identifying the major architectural patterns, particularly a reference architecture, if one is appropriate.
- Identifying infrastructure elements where the elements from the architectural patterns will execute or store data (operational and analytical) and communicate with each other.

Furthermore, if a DevOps approach is desired, some iterations should be focused on designing for deployability (see [Chapter 6](#)) and additional tasks should also be performed, including:

- Establishing a testing strategy and setting up pre-production and production environments (see [section 12.1.4](#)).
- Setting up build automation infrastructures, including continuous integration and deployment pipelines which will automate the deployment of the artifacts to the execution environments.
- Setting up monitoring infrastructures, including dashboards that allow the system to be observed at runtime.

This is the *iteration 0 + spikes* approach depicted in [Figure 12.3c](#). This will help to structure the project, define work assignments and team formation, address the most critical quality attributes, and prepare for early and continuous delivery, if needed. If and when requirements change—particularly if these are driving quality attribute requirements—adopt a practice of Agile experimentation, where spikes are used to address new requirements. A *spike* is a time-boxed task that is created to answer a technical question or gather information associated with important architectural decisions; it is not intended to lead to a finished product. Spikes are developed in a separate branch and, if successful, merged into the main branch of the code. In this way, emerging requirements can be welcomed and managed without being too disruptive to the overall process of development.

Agile architecture practices, however, help to tame some of the complexity, narrowing the cone of uncertainty and hence reducing project risk. A reference architecture defines families of technology components and their relationships. It guides integration and indicates where abstraction should be built into the architecture, to help reduce rework when a new technology (from within a family) replaces an existing one. Agile spikes allow prototypes to be built quickly and to “fail fast,” thereby guiding the eventual selection of technologies to be included on the main development branch.

12.1.2.2 Tension between emergence and planning

In an organizational context where multiple teams develop different products and reuse composable assets, it is highly unlikely to have a successful architecture simply “emerge”, particularly if multiple products share common assets.

Models that support scaling agile approaches, such as the Scaled Agile Framework (SAFe), consider the existence of two types of architectures, one that is “intentional” and one that is “emergent”. SAFe describes them in the following way:

- *Intentional architecture*: Defines a set of purposeful, planned architectural strategies and initiatives, which enhance solution design, performance, and usability and provide guidance for inter-team design and implementation synchronization.
- *Emergent design*: Provides the technical basis for a fully evolutionary and incremental implementation approach. This helps developers and designers respond to immediate user needs, allowing the design to evolve as the system is built and deployed.

In a context of business agility (see [Chapter 5](#)), there is a constant tension between emergence and planning in terms of architectural design. As the different products continuously evolve, potentially pivoting from the original vision as new features are added to the backlog, new architectural decisions potentially need to be made. Consider, for example, a product that was originally conceived for a national market which at some point in time must cater to an international audience. This product may not have been originally designed with internationalization or with a high volume of users in mind, and new design decisions need to be made to support these quality attribute scenarios. In such a situation, parts of the architecture may emerge, as the product’s original architecture is adjusted to support a higher volume of users and multiple user interfaces.

This product, however, may reuse shared common concerns and components which must be designed in ways that

satisfy quality attributes that are important not just to this specific product, but potentially to many other products developed in the organization. Examples of these quality attributes and concerns include security (such as rules and services for authentication or authorization), internationalization, logging, or error management. Changes in the design decisions that are common to multiple reusable assets can have a high impact in terms of the effort required to achieve them. For these types of assets, emergent architectural decisions can be very costly and thus it is preferable to make these costly design decisions intentionally and early on before many reusable composable assets are built. Planning promotes risk management and a mitigation mindset; for this reason, important architectural decisions that can have an important impact should be made early and not left to emerge at a later time. There is constant tension for the architect to decide how much generality to build into architectural decisions, and a balance must be found between more immediate and longer-term decisions.

12.1.2.3 Continuous architecture design and refactoring

In the context of business agility and product orientation, systems evolve continuously until they are retired. The backlog for these products changes constantly, with the addition of new features, but also with the addition of new architectural drivers such as the need for supporting a new quality attribute requirement or addressing an architectural concern.

This implies that architectural design is an activity that is performed continuously as the product evolves. It does not mean, however, that at every sprint the architecture will change, but rather that changes should be expected and

welcome at any time. Furthermore, refactoring is an activity that is performed continuously as changes in the product potentially result in technical debt that needs to be repaid. Tasks associated with architectural design and with technical debt should be part of the product backlog, as discussed in [section 10.2](#).

12.1.3 Designing to support sequential development

Certain projects are still developed using a sequential phase approach, although this approach is not nearly as widely used as in the past. While not a waterfall model, the Rational Unified Process (RUP) proposed that development projects should be divided into four major phases, which are carried out sequentially; within these phases, a number of iterations are performed. The four phases of the RUP are as follows:

1. *Inception.* In this first phase, the goal is to achieve concurrence among project stakeholders. During this phase the scope of the project and a business architecture are defined. Also, a candidate architecture is established. This phase is equivalent to the estimation phase discussed previously.
2. *Elaboration.* In the second phase, the goal is to baseline the architecture of the system and to produce architectural prototypes. It should be noted that RUP has a strong emphasis on architecture.
3. *Construction.* In the third phase, the goal is to incrementally develop the system from the architecture that was defined in the previous phase.
4. *Transition.* In the fourth phase, the goal is to ensure that the system is ready for delivery. The system is

transitioned from the development environment to its final operational environment.

We could argue that, from the elaboration phase until the end of the project, RUP intrinsically follows the iteration 0 + spikes approach described earlier. RUP also provides some guidance with respect to architectural design, although this guidance is far less detailed than that offered by ADD. Consequently, ADD can be used as a complement to the RUP. ADD iterations can be performed during inception to establish the candidate architecture by following the approach described in [Section 12.1.1](#). Furthermore, during the elaboration phase, the initial architecture is taken as a starting point for performing additional design iterations until an architecture that can be baselined is produced. During construction, additional ADD iterations may be performed as part of the development iterations.

A sequential phase approach also occurs in organizations that are transitioning from sequential development approaches to more agile ones. During the transition period, the organization performs what is often referred to as Water-Scrum-Fall development. In this context, there is a traditional planning phase followed by a development phase which is performed using Scrum. Once the development team finishes a sprint, the product increment is passed on to a QA team for testing. After tests are concluded, corrections are made and user stories are accepted, the system is passed to an operations team for moving to production. Although this type of development is not conducive to agility because it involves relatively static requirements and a slow delivery, it can still benefit from ADD. Similar to what was discussed about RUP, initial ADD iterations can be performed in the planning phase to produce an initial architecture, and an iteration 0 + spikes approach can be followed in the subsequent development.

We recommend teams or organizations that are stuck in the Water-Scrum-Fall approach to introduce DevOps practices and architectural support for these practices, to reduce the time needed to move the system from development to production and achieve a more agile approach.

12.1.4 Architecture design and the testing strategy

One important aspect of software development is planning how testing for a given system will be performed.

Depending on the type of system that is being developed, a specific testing strategy needs to be established.

The architecture of a software system has a direct impact on how the system can be tested. This is particularly salient for complex distributed systems where many components (such as microservices) compose an application and where these components depend on middleware resources, databases and external third-party systems. Establishing an effective testing strategy requires an architecture to be defined and must involve architects, quality engineers and people who manage infrastructure.

Several concerns must be considered when developing the testing strategy, including:

- *What is the number of pre-production environments that are expected?* As we saw in [section 6.1.1](#), there may be a number of pre-production environments where the system is integrated and tested before it is released to a production environment.
- *How are dependencies handled in the pre-production environments?* To function, the system may require the presence of shared reusable services, middleware components, databases and third-party systems. In an

integration environment, some of these dependencies may be substituted by mocks, however, in a staging environment, it may be necessary to have access to testing versions of these components (for example, a sandbox of a third-party system). In the case of shared reusable services, such as the ones that are present when the application is built on top of an API platform, additional difficulties arise. These are associated with ensuring that tests are conducted using the correct versions of the services that are reused.

- *How is infrastructure replicated across environments?* A complex system may involve many moving parts, and it may be necessary to replicate its infrastructure across pre-production and production environments. Infrastructure as code can be very helpful here as an infrastructure descriptor can be used to easily replicate the infrastructure across environments.
- *How to reduce costs?* Replicating the infrastructure across pre-production environments can be costly, especially in cloud environments. Some approaches to reduce costs can be to request more limited cloud resources in integration environments (e.g. non replicated resources, smaller DBs) and turning off environments during non-testing times. This can be challenging when dealing with shared services such as the ones that exist in API platforms as a shared service may be used by multiple teams.

Even relatively simple systems, such as the one discussed in the Hotel Pricing system case study in [chapter 8](#) present the challenges discussed here. For instance, this system is composed of four microservices with two different types of databases and a messaging system, besides a number of other components to support the frontend. Furthermore, the system also interacts with a number of external systems.

For this system, the testing strategy involved the use of limited resources and mocks in the integration environment (including a version of Kafka and databases deployed in a container). This strategy limited the possibility of using Infrastructure as Code to set up the integration environment, but it reduced costs. Infrastructure as Code was, however, used to set up the staging and production environments.

12.2 Architecture Design and the Organization

In 1967, Melvin Conway wrote a paper in which he stated an observation, and this observation has become widely known as Conway's Law. This "law" states that "Organizations which design systems (defined broadly) are constrained to produce designs which are copies of the communication structures of these organizations". Colloquially, this means that the structure of a software system will reflect the structure of the organization that designed it. For example, a colleague of ours reported on an architectural analysis that he did where there were three separate databases in the system. Why three databases you ask? Because there were three major contractors working on this large system and each of them had a database team, and they found something for these database teams to do! This was likely a sub-optimal architecture, but it was driven by Conway's Law, not by technical merit.

In the following subsections we discuss aspects related to the organization that have a direct impact on the way that architectures are designed.

12.2.1 Designing as an Individual or as a Team

In large and complex projects, it seems straightforward that an architecture team should be responsible for performing the design. Even in smaller projects, however, you may find that having more than one person participate in the design process yields important advantages. You can decide if only one person is the architect and the others are observers (as in the practice of pair programming) or if the group actively collaborates on design decisions (although even here we recommend that you have one lead architect).

There are various benefits from this approach:

- Two (or more) heads can be better than one, particularly if the design problem that you are trying to solve is different from ones that you have addressed before.
- Different people can have different areas of expertise that are useful in the design of the architecture, as we will discuss in the next section.
- Design decisions are reflected upon and reviewed as they are being made and, as a consequence, can be corrected immediately.
- It is well known that individuals have cognitive biases, and they are unaware of these biases. Having multiple designers and reviewers means that such biases are more likely to be flagged.
- Less experienced people can participate in the design process, which can be an excellent mentoring practice.

You should, however, be aware of certain difficulties with this approach:

- Design by committee can be complicated if agreement is not achieved in a reasonable time frame. The search for consensus can lead to “analysis paralysis.”
- The cost of design increases and, in many cases, the time for design also increases.
- Managing the logistics can be complex, because this approach requires the regular availability of the group of people.
- You may encounter personality and political conflicts, resulting in resentment or hurt feelings or in design decisions being heavily influenced by the person who shouts longest and loudest (“design by bullying”).

12.2.2 The Many Roles of the Architect

For most of this book we have mentioned the role of architect as the primary actor in the design activities. In smaller projects, a single person may fulfill this role and make design decisions across different fronts. In larger organizations, however, the role of architect may be split across many people, each of whom has a different area of expertise. This makes sense since it can be difficult for a single person to have expertise across all the areas where design decisions need to be made, especially on large and complex systems (see [Chapter 9](#) for a good example of this). Here we review some of the most common specializations of the architect role.

12.2.2.1 Software Architect

The role of the software architect is focused on making design decisions at the software level. They are responsible for defining the structure of the software system, including

its modules and their relationships—principally interfaces—to ensure that drivers are satisfied. The design of the software architecture also frequently involves the selection of technologies (i.e. externally developed components). The activities performed by this main role are, of course, discussed throughout this book.

12.2.2.2 Infrastructure Architect

Infrastructure architects are focused on designing the underlying infrastructure that supports software systems. They are responsible for defining the physical or virtual infrastructure components, such as servers, networks, storage, and cloud services, and ensuring that they are selected and configured to meet the performance, security, and scalability requirements of the system. Cloud architects are one type of infrastructure architects who specialize in the design of cloud-based infrastructures. They frequently specialize on the offerings of a particular cloud provider and are knowledgeable about the resources that the cloud provider offers and how they can be used together. When following an Infrastructure as Code approach, this role may participate not only in the design decisions but also in the selection of technologies to support this approach and, possibly, the definition of source code files where the infrastructure is described.

As we have discussed elsewhere in this book (such as [Chapter 6](#) on Deployability and [Chapter 7](#) on Cloud-based solutions), quality attributes frequently require a combination of design decisions at the software and the infrastructure level, so when different people occupy the roles of software and infrastructure architects, they need to collaborate closely to ensure that the software and the infrastructure work together optimally.

12.2.2.3 Security Architect

Security is a quality attribute of utmost importance in many of today's systems. It is also a quality attribute that requires extensive knowledge in many different areas including the types of threats and vulnerabilities that can occur in a system and the different design concepts including tactics, patterns and externally developed components that can be used to address this quality attribute. Security architects must be experts on these topics as they are responsible for designing and implementing secure information systems and infrastructures. Their role is to identify potential security risks, threats, and vulnerabilities and to design and implement appropriate security controls and solutions to mitigate those risks.

12.2.2.4 Data Architect

Design decisions about data are the realm of the Data Architect. Data architects are responsible for designing mechanisms where data is stored and retrieved securely, reliably, and efficiently. They work closely with business analysts, data scientists, and other stakeholders to understand the organization's data needs and to develop solutions that meet those needs. For instance, when designing enterprise applications, decisions about storage and management of operational versus analytical data must be made. Important aspects that need to be considered include the establishment of a data model and the lifecycle of its abstractions, the management of metadata, and the organization of data in different types of data stores, securing access to the data, retention policies, and so forth.

12.2.2.5 Other Architect Roles

Many areas in software development require design decisions to be made so, arguably, architecture design is

performed in a number of additional domains besides the ones that we have already mentioned. An example of this is the design of the infrastructure required to automate deployments, as this involves making design decisions on aspects such as the selection of technologies for the execution of pipelines along with the definition of the steps of the pipeline itself. As with any other architectural decisions, there are drivers associated with the design of this infrastructure. For example, it has to map to the repository branching model (e.g. Gitflow) that is used in the organization. There may be performance requirements regarding the time it takes to execute the pipeline and also constraints to, for example, not use proprietary solutions.

Another example of a specialized architecture role would be that of a blockchain architect. Similar to other architect roles, this is a person who must have very deep expertise but in this case on matters related to blockchain. Design decisions made by this role include aspects such as the selection of the type of blockchain network, the types of consensus mechanism to be used, the type of smart contracts and the specific technologies to be used.

Another type of specialized architect is AI or Machine Learning architects who, in a similar vein to the two examples previously discussed, are experts on making design decisions in their field, about the algorithms and learning methods selected, online versus offline learning, centralized versus distributed approaches, model retraining frequencies, and so forth.

While the previous examples of specialized architect roles are on a similar level in the organizational chart, some organizations or models also have vertical hierarchies of architects. In this hierarchy, enterprise architects are typically “above” the software architects. While software architects are typically focused on the design of a particular

system, enterprise architects are focused on making design decisions at the organizational level (i.e. the intentional architecture). They are responsible for designing and managing the overall architecture of an organization's IT systems, standards, processes, and technologies. They work to align the organization's technology strategy with its overall business objectives and help to ensure that all IT systems are integrated, efficient, and effective. The enterprise architecture group generally establishes a governance body that provides guidance to the activities performed by the other architects. Therefore, they define constraints (in the context of the architectural design drivers) which should be considered by the other architect roles like software and infrastructure architects.

12.2.2.6 Architect Roles and ADD

Integrating different architect roles into ADD can be performed by considering the goals of specific design iterations (step 2 of ADD). For example, an iteration whose goal is a driver focused on producing an initial structure of the system may only require the expertise of a software architect. However, an iteration whose goal is a driver associated with creating an efficient catalog of products for a large e-commerce system that will be hosted in the cloud may involve not only a software architect but also a cloud and a data architect, and perhaps even an enterprise architect. Each architect role plays an important part in the key design steps of ADD (steps 3 to 5), as each one of them is an expert on the design concepts associated with their particular area and also in instantiating these design concepts to address the driver that is the goal of the iteration.

These architect roles must collaborate since a decision made by the software architect to store information may be

complemented by a recommendation from the data architect regarding the type of database to be used. This, in turn, must be validated with the cloud architect to discuss the database capabilities that the cloud provider offers to see which one is the best fit. The software architect may perform the design iteration and consult with the other architects when their point of view is required, or all the architects may participate in performing the key design steps. Considering what we mentioned in [section 12.2.1](#), while the design of the system may be performed by a team of experts, one person should be responsible for *leading* the design process and its iterations; ideally, that is the responsibility of the software architect.

12.2.3 Architecture guidelines for the organization

In larger organizations, there are a number of different teams that work on the development of applications or a platform that supports these applications (such as the API platform we presented in [Chapter 5](#)). Within such organizations it is often desirable that teams solve certain design aspects in a common way. Here we discuss some approaches that are helpful to achieve this.

12.2.3.1 Addressing specific architectural concerns

While teams are usually focused on solving specific design problems associated with their particular projects, there are some common design aspects that all teams must address. These aspects are typically associated with making decisions associated with the architectural concerns that we discussed in [section 2.4.4](#), and they involve considerations such as:

- The selection of a deployment approach (monolithic vs non-monolithic).
- The selection of mechanisms and standards for communication.
- The selection of security mechanisms to address aspects such as authorization and authentication, and encryption of data.
- The selection of an approach for versioning APIs.
- The selection of mechanisms for logging both for auditing and for debugging.
- The selection of mechanisms to support observability and debugging, including error management and distributed tracing.
- The selection of appropriate data storage options (e.g. how long data needs to be stored), considering aspects such as compression and encryption.
- The selection of mechanisms to support internationalization including the management of time, currencies, and languages.
- The selection of mechanisms to support privacy, including protecting sensitive information from exposure and preparation of data for testing environments.
- The selection of development tools and platforms including programming languages, repositories, coding conventions and standards.

Within an organization it is frequently desirable to establish uniform mechanisms to address a number of these specific concerns. A good example of this is logging. If every team decides on a particular way to address logging, there may be issues if the company decides to centralize the storage, analysis, and comparison of logs. Instead, the organization

can provide guidelines and mechanisms, such as libraries or services to handle this specific architectural concern which facilitates the centralized approach for storage and analysis. The definition of these guidelines and common design decisions to address these architectural concerns can be performed by a governance body such as the enterprise architecture group.

12.2.3.2 Design Concepts Catalogs

While the guidelines discussed in the previous section focus on addressing common architectural concerns, it may also be useful to an organization to provide support for identifying design concepts to solve recurring problems.

As we saw in [Section 4.4](#), the selection of concepts is one of the most challenging aspects of the design process. This problem is exacerbated by the fact that information is scattered across many locations: architects usually need to consult several pattern and tactics catalogs and do extensive research to find the design concepts that can be considered and used.

One possible way to resolve this issue is the creation of *design concept catalogs*. These catalogs group collections of design concepts for particular application domains. Such catalogs are intended to facilitate the identification and selection of concepts when performing design. They are also useful in enhancing consistency in the designs across the organization. For example, designers may be required to use the technologies in a particular catalog as much as possible because this facilitates estimation, reduces learning curves, reduces costs and risks, facilitates evaluating proposed architectures, and may lead to opportunities for reuse. Catalogs can also be useful for training purposes.

An example of a design concept catalog can be found online in the companion site to this book.

The creation of these catalogs involves considerable effort and, once created, they should be maintained as new design concepts, and particularly new technologies, are introduced or removed in the organization. This effort is worthwhile, however, as these catalogs are a valuable organizational asset.

12.2.4 Architecture groups

Large companies frequently employ a number of architects. When that situation exists, it is desirable to establish architecture groups that promote the collaboration of architects in different activities such as:

- *Sharing knowledge*: architects should be incentivized to present new approaches and tools to the group, for learning purposes or for evaluating whether to incorporate them into the set of technologies used in the company. Contributing to the design concepts catalog of the company, if it exists, is another effective form of sharing knowledge.
- *Discussing particular issues*: architects can present particular design challenges or solutions to the group to obtain feedback or opinions from other points of view. This can also highlight opportunities for sharing and reuse.
- *Evaluating proposed solutions*: In [section 11.6](#) we discussed an analysis technique called “Scenario-based reviews” and the steps to perform it. This technique requires a team of evaluators that reviews and questions the design to uncover risks. Other architects

in the company are frequently ideal candidates to participate in this evaluation team.

These activities can be performed periodically, for example through a weekly architecture group meeting, although architectural evaluations usually require a dedicated meeting.

12.3 Summary

In this chapter we discussed how ADD can be used in relation to several organizational aspects. ADD can be used from the project's inception to facilitate estimation using standard components. As the project evolves, ADD can be used in conjunction with both modern and traditional software development life-cycle methods. In general, ADD is a valuable complement to life-cycle methods that do not provide detailed guidance on how to perform architectural design.

We also discussed aspects related to supporting architectural design at the organizational level, including the composition of the design team, the different architect roles, providing guidelines at the organizational level to address architectural concerns and to identify design concepts using catalogs. Finally, we discussed the benefits of creating architecture groups in organizations where there are a number of architects in the IT department.

12.4 Further Reading

Organizational structure and its influences on software architecture are addressed in the field of enterprise architecture management. Enterprise architecture frameworks are discussed in F. Ahlemann et al. (Eds.), *Strategic Enterprise Architecture Management*:

Challenges, Best Practices, and Future Developments,
Springer-Verlag Berlin Heidelberg, 2012.

The book “Architecture Centric Project Management: A Practical Guide” by D. Paulish discusses many aspects of project management and connects them to software architecture.

The definitions of Intentional Architecture and Emergent Design are © by Scaled Agile, Inc. Additional details about the Scaled Agile Framework can be found online at: <https://scaledagileframework.com/>

The concept of the cone of uncertainty has existed in the project management domain for decades. Barry Boehm employed this concept to software projects in his seminar work *Software Engineering Economics*, Prentice Hall, 1981.

A nice set of articles looking at the relationship between architecture and Agile methods can be found in the April 2010 *IEEE Software* magazine special issue on this topic.

A number of studies have looked at how architecture and agility methods complement and support each other, such as S. Bellomo, I. Gorton, and R. Kazman, “Insights from 15 Years of ATAM Data: Towards Agile Architecture”, *IEEE Software*, 2015, and S. Bellomo, R. Nord, and I. Ozkaya, “A Study of Enabling Factors for Rapid Fielding: Combined Practices to Balance Speed and Stability”, *Proceedings of ICSE 2013*, 982–991, 2013.

Barry Boehm and Richard Turner have taken an empirical look at the topic of the relationship between agility and “discipline” (not just architecture) in their book *Balancing Agility and Discipline: A Guide for the Perplexed* (Boston: Addison-Wesley, 2004).

The practice of creating architectural “spikes” as a means of resolving uncertainty in Agile sprints is discussed in T. C.

N. Graham, R. Kazman, and C. Walmsley, "Agility and Experimentation: Practical Techniques for Resolving Architectural Tradeoffs", *Proceedings of the 29th International Conference on Software Engineering (ICSE 29)*, (Minneapolis, MN), May 2007. A general discussion of spikes can be found at
<https://www.scrumalliance.org/community/articles/2013/march/spikes-and-the-effort-to-grief-ratio>

Many practitioners and researchers have thought deeply about how Agile methods and architectural practices fit together. Some of the best examples of this thinking can be found in the following sources:

- S. Brown. *Software Architecture for the Developers*. LeanPub, 2013
- J. Bloomberg. *The Agile Architecture Revolution*. Wiley CIO, 2013.
- A. Cockburn. "Walking Skeleton".
<http://alistair.cockburn.us/Walking+skeleton>
- "Manifesto for Agile Software Development".
<http://agilemanifesto.org/>
- Scott Ambler and Mark Lines. "Scaling Agile Software Development: Disciplined Agility at Scale".
<http://disciplinedagileconsortium.org/Resources/Documents/ScalingAgileSoftwareDevelopment.pdf>
- The Agile Architecting Collection in the SEI Digital Library: <https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=483941>

An extensive treatment of estimation techniques, including estimation using standard components, is given in S. McConnell, *Software Estimation: Demystifying the Black Art*, Microsoft Press, 2006.

The integration of ADD 2.0 (as well as other architecture development methods) with RUP, is discussed in R. Kazman, P. Kruchten, R. Nord, and J. Tomayko, “Integrating Software-Architecture-Centric Methods into the Rational Unified Process”, Technical Report CMU/SEI-2004-TR-011, July 2004.

Considerable attention has been given to the problem of architecture knowledge representation and management. For a good overview of this area, see P. Kruchten, P. Lago, and H. Van Vliet, “Building Up and Reasoning About Architectural Knowledge”, in *Quality of Software Architectures*, Springer, 2006. For a perspective on tools for architecture knowledge management, see A. Tang, P. Avgeriou, A. Jansen, R. Capilla, and M. Ali Babar, “A Comparative Study of Architecture Knowledge Management Tools”, *Journal of Systems and Software*, 83(3):352–370, 2010.

12.5 Discussion questions

1. What additional actions relative to architectural design would you suggest to undertake in the estimation phase to further reduce the uncertainty in estimates? Consider that there are contexts where the work performed in this phase is not compensated and will only be compensated when the project is approved.
2. Considering an Iteration 0 + spikes approach, what can happen if the team doesn't devote time to design for deployability (that is, they do not design and set up the execution environments and build automation infrastructures)? Once the project is running how can they remedy this situation?
3. What other architect roles can you identify besides the ones that are mentioned in [section 12.2.2](#)?

4. Considering the role of an AI architect, what kind of drivers and design concepts can be associated with this particular architect role? Provide examples.
5. What problems can happen in an organization where multiple teams develop APIs but there are no company-wide guidelines on how to address authorization and authentication, versioning or error management?
6. Similarly, what problems can occur in an organization that develops microservices, but there are no company-wide guidelines on what microservices to create, at what granularity, accessing what data?
7. Given Conway's Law, it is obvious that an architect needs to "shepherd" both the technical and social dimensions of a project. What are some techniques for monitoring a project's socio-technical alignment, and what are some techniques for remedying misalignments?

13. Final Thoughts

In this chapter we reflect, once again, on the nature of design and why we need methods for design. This is, after all, the major point of this book! And we leave you with a few words about where to go with the information and skills that you have gleaned from reading this book.

13.1 On the Need for Methods

Given that you have prevailed and reached this final chapter, we can assume that you are committed to being a *professional* software architect. Being a professional means that you can perform consistently at a high level in all sorts of business and technical contexts. To achieve this level of consistent performance, everyone needs *methods*. This is why standardized methods are employed in manufacturing, science, healthcare, education, IT, and many other domains.

We all need methods when we are performing complex tasks that have serious consequences if we get them wrong. Consider this: Jet pilots and surgeons are two of the most highly trained groups of professionals in the world, and yet they use checklists and standardized procedures for every important task that they perform. Why? Because the consequences of making a mistake are serious. You

probably will not be designing the architectures for systems that have life-and-death consequences. Even so, the systems that you do design, particularly if they are large and complex, may very well have consequences for the health and well-being of your organization. If you are designing a throwaway prototype or a trivial system, perhaps an explicit architecture design step may be omitted. If you are designing the *n*th variant of a system that you have created over and over in the past, perhaps architecture design is little more than a cut-and-paste from your prior experiences.

But if the system you are charged with creating or evolving is nontrivial and if there is *risk* associated with its creation, then you owe it to yourself, you owe it to your organization, and you owe it to your profession to do the best job that you can in this most critical step in the software development life cycle. To achieve that goal, you probably need a method. Methods help to ensure uniformity, consistency, and completeness. Methods help you take the right steps and ask the right questions.

Of course, no method can substitute for proper training and education. No one would trust a novice pilot at the controls of a 787 or a first-year medical student wielding a scalpel in an operating theater, armed only with a method or a checklist. A method, however, is a key to producing high-quality results repeatedly. And this is, after all, what we all desire as software engineering professionals.

Fred Books, writing about the design process, said:

Any systematization of the design process is a great step forward compared to “Let’s just start coding, or building.” It provides clear steps for planning a design project. It furnishes clearly definable milestones for planning a schedule and for judging progress. It suggests

project organization and staffing. It helps communication within the design team, giving everyone a single vocabulary for the activities. It wonderfully helps communication between the team and its manager, and between the manager and other stakeholders. It is readily teachable to novices. It tells novices facing their first design assignments where to begin.

Design is just too important to be left to chance. And there needs to be a better way of getting good at design than “shoot yourself in the foot repeatedly.” As the Nobel Prize-winning scientist Herbert Simon wrote in 1969, “Design...is the core of all professional training; it is the principal mark that distinguishes the professions from the sciences. Schools of engineering, as well as schools of architecture, business, education, law, and medicine, are all centrally concerned with the process of design.” Simon went on to say that lack of professional competence is caused by the relative neglect of design in universities’ curricula. This trend is, we are happy to note, gradually reversing, but over 50 years later we can not claim victory; this is still a cause for concern and attention.

In this book we have provided you with a road-tested method—ADD—for doing architectural design. Methods are useful in that they provide guidance for the novice and reassurance for the expert. Like any good method, ADD has a set of steps. And while ADD has evolved over the years, the current set of steps has remained stable for the past decade, even as our focus has broadened from single systems deployed on your own hardware to collections of services deployed on the cloud, from big-bang deployments to continuous delivery, from design of modules to design of microservices and APIs.

But just as important, we have focused on the broader architecture life cycle and shown how some changes to the

design process can help make your life as an architect better, and provide you with better outcomes. For example, we have expanded the set of inputs that you need to think about to include things like design purpose and architectural concerns. This broader view helps you create an architecture that not only meets your customer's requirements, but also is aligned with the business needs of your team and your organization. In addition, we have shown that design can and should be guided by a "design concepts catalog" —a corpus of reusable architectural knowledge consisting of reference architectures, patterns, tactics, and externally developed components such as frameworks and technology families. By cataloging these concepts, design can be made more predictable, efficient, and repeatable. Finally, we have argued that design should be documented, perhaps informally in sketches with accompanying rationale, and should be accompanied by a consistent practice of analyzing the decisions made.

If we are to conceive of ourselves as software engineers, we need to take the title of "engineer" seriously. No mechanical or electrical or structural engineer would commit significant resources to a design that was not based on sound principles and components, or that was not analyzed and documented. We think that software engineering in general, and software architecture specifically, should strive for similar goals. We are not "artistes," for whom creativity is paramount; we are engineers, so predictability and repeatability should be our most cherished goal.

13.2 Future Directions

While technologies have evolved significantly from the first edition of the book, the principles of software architecture haven't changed. The process of designing an architecture remains the same and, as we mentioned in [chapter 4](#), we

did not modify ADD from the version that we published in the first edition of the book. While this gives us comfort--knowing that design is design and how we do it does not depend on the latest technologies, recent developments in artificial intelligence may actually have an impact on the way we architect in the future. Specialized Large Language Models (LLMs) may be useful in several areas associated with software architecture including, for design:

- Generating or analyzing requirements and architectural drivers.
- Assisting in the selection of design concepts and their instantiation.
- Automating the generation of diagrams and documentation.
- Generating test cases associated with the architectural design.
- and for refactoring and maintenance:
- Explaining the concepts employed in existing code.
- Suggesting code changes to remove bad smells.
- Updating documentation to match what is actually implemented.

As we write this book, it is still early to perform these tasks effectively and with high confidence. However, small proofs of concept are already feasible and developers the world over are embracing LLMs. The future use of this technology is exciting as LLMs can be based on a broader knowledge base, and have less bias than humans. Also, these tools can free up architects and developers to focus on the more interesting and challenging aspects of design, rather than the mundane stuff (like documentation) that people often

resist or avoid. We will surely witness exciting developments around this topic in the coming years.

13.3 Next Steps

Where should you go from here? We see four answers to this question. One answer focuses on what you can do as an individual to hone your skills and experience as an architect. The second answer revolves around how you might engage your colleagues to think more consciously about architecture design. The third answer is where your organization can go with a more explicit commitment to architecture design. And the fourth answer is about how you can contribute to your community, and to the larger community of software architects.

Our advice to you, as an individual, about how to proceed is simple: *practice*. Like any other complex skill worth having, your skill as an architect will not come immediately, but your confidence should increase steadily. “Fake it till you make it” is the best advice that we can give. Having a method that you can consult, and a ready supply of common design concepts, gives you a solid foundation on which to “fake it” and learn.

To help you practice your skills and to engage your colleagues, we have developed an architecture game. This game, which is called “Smart Decisions,” can be found at <http://www.smartdecisionsgame.com>. It simulates the architecture design process using ADD and promotes learning about it in a fun, pressure-free way. The game is focused on a specific application domain, but it can be easily adapted to other application domains.

You might also think about next steps to be taken in your organization. You can be an agent for change. Even if your company does not “believe in” architecture, you can still

practice many of the ideas embodied in this book and in ADD. Ensure that your quality attribute requirements are clear by insisting on concrete response measures. Even when facing tight deadlines and schedule pressures, try to get agreement on the major architectural patterns being employed. Do periodic, quick, informal design reviews with colleagues, huddled around a whiteboard, and ask yourself reflective questions. None of these “next steps” needs to be daunting or hugely time-consuming. And we believe—and our industrial experience has shown—that they will be self-reinforcing. Better designs will lead to better outcomes, which will lead you and your group and your organization to want to do more of the same.

Finally, you can contribute to your local software engineering community, and even to the worldwide community of software architects. You could, for example, play the architecture game in a local software engineering meetup and then share your experiences. You could contribute case studies about your successes and failures as an architect with real-world projects. We strongly believe that example is the best way to teach and while we have provided two case studies in this book, more is always better.

Happy architecting!

13.4 Further Reading

The long quotation by Fred Books in this chapter comes from his thought-provoking book *The Design of Design: Essays from a Computer Scientist*, Pearson, 2013.

Many of the ideas in this chapter, in this book, and in the field of software architecture in general can be traced back to Herbert Simon’s seminal book on the science of design: *The Sciences of the Artificial*, MIT Press, 1969.

13.5 Discussion questions

1. Experienced architects are sometimes reluctant to accept the idea that design can be performed using a systematic method. Why do you think this is the case? What are the benefits from using a method such as ADD?
2. Can you think of other areas of software architecture where Large Language Models can be used?
3. What benefits or drawbacks can you identify of using Artificial Intelligence tools to assist in the design process?
4. As architects and developers rely more on existing frameworks and components, especially the ones from cloud providers, does their role get more or less important?

Appendix A. Tactics-Based Questionnaires

This appendix provides a set of tactics-based questionnaires for the ten most important and widely used quality attributes: availability, deployability, energy efficiency, interoperability, modifiability, performance, safety, security, testability, and usability.

These questionnaires could be used by an analyst, who poses each question, in turn, to the architect and records the responses, as a means of conducting a lightweight architecture review. Alternatively, the questionnaires could be employed as a set of reflective questions that you could, on your own, use to examine your architectural choices.

In either case, to use these questionnaires, simply follow these four steps:

1. For each tactics question, fill the “Supported” column with Y if the tactic is supported in the architecture and with N otherwise. The tactic name in the “Tactics Question” column appears in **bold**.
2. If the answer in the “Supported” column is Y, then in the “Design Decisions and Location” column describe the specific design decisions made to support the tactic

and enumerate where these decisions are manifested (located) in the architecture. For example, indicate which code modules, frameworks, or packages implement this tactic.

3. In the “Risk” column, indicate the anticipated/experienced difficulty or risk of implementing the tactic using a (H = high, M = medium, L = low) scale. For example, a tactic that was of medium difficulty or risk to implement (or which is anticipated to be of medium difficulty, if it has not yet been implemented) would be labeled M.
4. In the “Rationale” column, describe the rationale for the design decisions made (including a decision to *not* use this tactic). Briefly explain the implications of this decision. For example, you might explain the rationale and implications of the decision in terms of the effort on cost, schedule, evolution, and so forth.

A.1 Availability

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Detect faults	Does the system use ping/echo to detect a failure of a component or connection, or network congestion?				
	Does the system use a component to monitor the state of health of other parts of the system? A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack.				

	Does the system use a heartbeat —a periodic message exchange between a system monitor and a process—to detect a failure of a component or connection, or network congestion?				
	Does the system use a timestamp to detect incorrect sequences of events in distributed systems?				
	Does the system do any sanity checking : checking the validity or reasonableness of a component's operations or outputs?				
	Does the system do condition monitoring , checking conditions in a process or device, or validating assumptions made during the design?				
	Does the system use voting to check that				

replicated components are producing the same results? The replicated components may be identical replicas, functionally redundant, or analytically redundant.

	<p>Do you use exception detection to detect a system condition that alters the normal flow of execution (e.g., system exception, parameter fence, parameter typing, timeout)?</p>				
	<p>Can the system do a self-test to test itself for correct operation?</p>				
Recover from faults (preparation and repair)	<p>Does the system employ redundant spares? Is a component's role as active versus spare fixed, or does it change in the presence of a fault? What is the switchover mechanism? What is the trigger for a switchover? How long does it take for a spare to assume its duties?</p>				
	<p>Does the system employ exception handling to deal with faults? Typically the handling involves either</p>				

	reporting the fault or handling it, potentially masking the fault by correcting the cause of the exception and retrying.				
	Does the system employ rollback , so				

	that it can revert to a previously saved good state (the “rollback line”) in the event of a fault?			
	Can the system perform in-service software upgrades to executable code images in a non-service-affecting manner?			
	Does the system systematically retry in cases where the component or connection failure may be transient?			
	Can the system simply ignore faulty behavior (e.g., ignore messages sent from a source when it is determined that those messages are spurious)?			
	Does the system have a policy of degradation when resources are compromised, maintaining the most <small>critical systems</small> ?			

	<p>critical system functions in the presence of component failures, and dropping less critical functions?</p>				
	<p>Does the system have consistent policies and mechanisms for</p>				

	<p>reconfiguration after failures, reassigning responsibilities to the resources left functioning, while maintaining as much functionality as possible?</p>			
Recover from faults (reintroduction)	<p>Can the system operate a previously failed or in-service upgraded component in a “shadow mode” for a predefined time prior to reverting the component back to an active role?</p>			
	<p>If the system uses active or passive redundancy, does it also employ state resynchronization, to send state information from active to standby components?</p>			
	<p>Does the system employ escalating restart—that is, does it recover from faults by varying the granularity of the component(s)</p>			

restarted and minimizing the level of service affected?				
Can message processing and routing portions of the system employ nonstop forwarding , where				

	functionality is split into supervisory and data planes? In this case, if a supervisor fails, a router continues forwarding packets along known routes while protocol information is recovered and validated.			
Prevent faults	Can the system remove components from service , temporarily placing a system component in an out-of-service state, for the purpose of mitigating potential system failures?			
	Does the system employ transactions —bundling state updates so that asynchronous messages exchanged between distributed components are <i>atomic, consistent, isolated, and durable</i> ?			
	Does the system use a predictive model to			

	monitor the state of health of a component to ensure that the system is operating within nominal parameters? When conditions are detected that are predictive of likely future faults, the model initiates corrective action.			
	Does the system prevent exceptions from occurring by, for example, masking a fault, using smart pointers, abstract data types, or wrappers?			
	Has the system been designed to increase its competence set , for example by designing a component to handle more cases—faults—as part of its normal operation?			

A.2 Deployability

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Manage Deployment Pipeline	Can the system scale rollouts of a new version of a service gradually, to subsets of the user base, rather than deploying to the entire user base at once?				
	Is there a standard and broadly used way to script deployment commands for the complex steps to be				

	carried out and precisely orchestrated in a deployment?			
	If a deployment has defects it can be rolled back to its prior state, in a fully automated fashion?			
Manage Deployed System	Does the system rigorously manage service interactions among deployed versions of interacting services so that incompatibilities are avoided?			
	Does the system package dependencies among elements so that they are deployed together with all dependencies required to execute?			
	Does the system implement feature toggle —a “kill switch” that can be used for new features to automatically disable them without forcing a			

	new deployment?			
	Does the system externalize configurations , avoiding any hardcoded configurations that limit the possibility of moving it from one environment to another?			

A.3 Energy Efficiency

Tactics Group	Tactics Question	Sup-ported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Resource Monitoring	<p>Does your system meter the use of energy?</p> <p>That is, does the system collect data about the actual energy consumption of computational devices, via a sensor infrastructure, in near real time?</p>				
	<p>Does the system statically classify devices and computational resources? That is, does the system have reference values to estimate the energy consumption of a device or resource (in cases where real-time metering is infeasible or too computationally expensive)?</p>				
	Does the system				

dynamically classify
devices and

	<p>computational resources? In cases where static classification is not accurate due to varying load or environmental conditions, does the system use dynamic models, based on prior data collected, to estimate the varying energy consumption of a device or resource at run-time?</p>			
Resource Allocation	<p>Does the system reduce usage to scale down resource usage? That is, can the system deactivate resources when demands no longer require them, to save energy? This may involve spinning down hard drives, darkening displays, turning off CPUs or servers, running CPUs at a slower clock rate, or shutting down memory blocks of the processor that are not being used.</p>			
	<p>Does the system</p> <p>.....</p>			

schedule resources to more effectively utilize energy, given task constraints and respecting task priorities, switching computational resources, such as service providers, to

	ones that offer better energy efficiency, or lower energy costs? Is scheduling based on data collected (using one or more <i>Resource Monitoring</i> tactics) about the state of the system?			
	Does the system make use of a discovery service to match service requests to service providers? In the context of energy efficiency, a service request could be annotated with energy requirement information, allowing the requestor to choose a service provider based on its (possibly dynamic) energy characteristics.			
Reduce Resource Demand	Do you consistently attempt to reduce resource demand ? Here, you may insert the questions in this category from the Tactics-based Questionnaire for			

Questionnaire for
Performance (Chapter
9).

A.4 Integrability

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Limit Dependencies	Does the system encapsulate functionality of each element by introducing explicit interfaces and requiring that all access to the elements passes through these interfaces?				
	Does the system broadly <i>use</i> intermediaries for breaking dependencies between components, for example, removing a data producer's knowledge of its consumers?				
	Does the system abstract common services , providing a general, abstract interface for similar services?				



	Does the system provide a means to restrict communication paths between components?				
	Does the system adhere to standards in terms of how components interact and share information with each other?				
Adapt	Does the system provide the ability to statically (i.e., at compile time) tailor interfaces , that is, the ability to add or hide capabilities of a component's interface without changing its API or implementation?				
	Does the system provide a discovery service , cataloguing and disseminating information about services.				

	Does the system provide a means to configure the behavior of components at build, initialization, or runtime?			
Coordinate	Does the system include an orchestration mechanism that coordinates and manages the invocation of components so they can be unaware of each other?			
	Does the system provide a resource manager that governs access to computing resources?			

A.5 Modifiability

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Increase cohesion	Do you make modules more cohesive by splitting the module ? For example, if you have a large, complex module can you split it into two (or more) more cohesive modules?				
	Do you make modules more cohesive by redistributing responsibilities ? For example, if responsibilities in a module do not serve the same purpose, they should be				

	placed in other modules.			
Reduce coupling	<p>Does the system consistently encapsulate functionality? This typically involves isolating the functionality under scrutiny and introducing an explicit interface to it.</p>			
	<p>Does the system consistently use an intermediary to keep modules from being too tightly coupled? For example, if A calls concrete functionality C, you might introduce an abstraction B that mediates between A and C.</p>			
	<p>Do you restrict dependencies between modules in a systematic way? Or is any system module free to interact with any other module?</p>			

	Does the system abstract common services , in cases where you are providing several similar services? For example, this technique is often used when you want your system to be portable across operating systems, hardware, or other environment variations.				
Defer binding	Does the system regularly defer binding of important functionality so that it can be replaced later in the life cycle, perhaps even by end users? For example, do you use plugins, add-ons, or user scripting to extend the functionality of the system?				

A.6 Performance

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions & Location	Rationale and Assumptions
Control resource demand	Can you manage work requests ? For example, do you have in place a Service Level Agreement (SLA) that specifies the maximum event arrival rate that you are willing to support? Can you manage the rate at which you sample events arriving at the system?				
	Does the system monitor and limit its event response ? Does the system limit the number of events it responds to in a time period, to ensure predictable response?				
	Given that you may have more requests for service than available resources, does the system prioritize events ?				
	Does the system reduce the overhead of responding to				

	service requests by, for example, removing intermediaries or co-locating resources?			
	Does the system monitor and bound execution time ? More generally, do you bound the amount of any resource (e.g., memory, CPU, storage, bandwidth, connections, locks) expended in response to requests for services?			
	Do you increase resource efficiency ? For example, do you regularly improve the efficiency of algorithms in critical areas, to decrease latency and improve throughput?			
Manage resources	Can you allocate more resources to the system or its components?			
	Are you employing concurrency ? If requests can be processed in parallel, the blocked time can be reduced.			

	Are there computations that can be replicated on different processors?			
	Is there data that can be cached (to maintain a local copy that can be quickly accessed), or replicated (to reduce contention)?			
	Can queue sizes be bounded to place an upper bound on the resources needed to process stimuli?			
	Have you ensured that the scheduling strategies you are using are appropriate for your performance concerns?			

A.7 Safety

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Unsafe state avoidance	Do you employ substitution , that is safer, often hardware-based protection mechanisms for potentially dangerous software design features?				
	Do you use a predictive model to predict the state of health of system processes, resources, or other properties—based on monitored information—to ensure that the system is not only operating within its nominal operating parameters, but also to provide early warning of a potential problem?				

Unsafe state detection	Do you use timeouts to determine whether the operation of a component meets its timing constraints?				
	Do you use timestamps to detect incorrect sequences of events?				
	Do you employ condition monitoring to check conditions in a process or device, particularly to validate assumptions made during design?				
	Is sanity checking employed to check the validity or reasonableness of specific operation results, or inputs or outputs of a component?				
	Does the system employ comparison to detect unsafe states, by comparing the outputs produced by a number of synchronized or replicated elements?				
Containment - Redundancy	Do you use replication —clones of a component—to protect against random failures of hardware?				

Do you use functional redundancy , to address the common-mode failures by implementing diversely-designed components?				
Do you use analytic redundancy —functional “replicas” that include high assurance/high performance and				

	low assurance/low performance alternatives—to be able to tolerate specification errors?			
Containment - Limit Consequences	Can the system abort an operation that is determined to be unsafe before it can cause damage?			
	Does the system provide controlled degradation , where the most critical system functions are maintained in the presence of component failures, dropping or degrading less critical functions?			
	Does the system mask a fault by comparing the results of several redundant components and employ a <i>voting</i> procedure in case one or more of the components differ?			
Containment - Barrier	Does the system support limiting access to critical resources (e.g. processors, memory, and network connections) through a firewall ?			

	Does the system control access to protected components and protect against failures arising from incorrect sequencing of events through interlocks ?			
Recovery	Is the system able to rollback , to revert to a previous known good state, upon the detection of a failure?			
	Can the system repair a state determined to be erroneous, without failure, and then continue execution?			
	Can the system reconfigure resources, in the event of failures, by re-mapping the logical architecture onto the resources left functioning?			

A.8 Security

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Detecting attacks	Does the system support the detection of intrusions ? An example is comparing network traffic or service request patterns within a system to a set of signatures or known patterns of malicious behavior stored in a database.				
	Does the system support the detection of denial-of-service attacks ? An example is the comparison of the pattern or signature of network traffic coming into a system to historic profiles of known denial-of-service attacks.				

	<p>Does the system support the verification of message integrity? An example is the use of techniques such as checksums or hash values to verify the integrity of messages, resource files, deployment files, and configuration files.</p>				
	<p>Does the system support the detection of message delays? An example is checking the time that it takes to deliver a message.</p>				
Resisting attacks	<p>Does the system support the identification of actors? An example is identifying the source of any external input to the system.</p>				
	<p>Does the system support the authentication of actors? An example is ensuring that an actor (a user or a remote computer) is actually</p>				

<p>computer is actually who or what it purports to be.</p>				
<p>Does the system support the authorization of actors? An example is ensuring that an authenticated actor has</p>				

<p>the rights to access and modify either data or services.</p>				
<p>Does the system support limiting access? An example is controlling what and who may access which parts of a system, such as processors, memory, and network connections.</p>				
<p>Does the system support limiting exposure? An example is reducing the probability of a successful attack, or restricting the amount of potential damage, by concealing facts about a system (“security by obscurity”) or dividing and distributing critical resources (“don’t put all your eggs in one basket”).</p>				
<p>Does the system support data encryption? An example is to apply</p>				

some form of encryption to data and to communication.				
Does the system design consider the separation of entities ? An example is the physical separation of different				

	servers attached to different networks, the use of virtual machines, or an “air gap.”			
	Does the system support changing credential settings ? An example is forcing the user to change settings assigned by default.			
	Does the system validate input in a consistent, system-wide way? An example is the use of a security framework or validation class to perform actions such as filtering, canonicalization, and escaping of external input.			
Reacting to attacks	Does the system support revoking access ? An example is limiting access to sensitive resources, even for normally legitimate users and uses, if an attack is suspected.			

Does the system support restricting login in instances such as multiple failed login attempts?				

	Does the system support informing actors ? An example is notifying operators, other personnel, or cooperating systems when an attack is suspected or detected.				
Recovering from attacks	Does the system support maintaining an audit trail ? An example is keeping a record of user and system actions and their effects, to help trace the actions of, and to identify, an attacker				
	Does the system guarantee the property of nonrepudiation , which guarantees that the sender of a message cannot later deny having sent the message and that the recipient cannot deny having received the message?				
	Have you checked the “recover from faults” category of tactics from				

	Category or feature from Availability?				
--	---	--	--	--	--

A.9 Testability

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Control and observe system state	Does the system or the system components provide specialized interfaces to facilitate testing and monitoring?				
	Does the system provide mechanisms that allow information that crosses an interface to be recorded so that it can be used later for testing purposes (record/playback)?				
	Is the state of the system, subsystem, or modules stored in a single place to facilitate testing (localized state storage)?				
	Can you abstract data sources —for example, by abstracting interfaces? Abstracting the interfaces lets you substitute test data				

	more easily.			
	Can the system be executed in isolation (a sandbox) to experiment or test it without worrying about having to undo the consequences of the experiment?			

	Are executable assertions used in the system code to indicate when and where a program is in a faulty state?				
Limit complexity	Is the system designed in such a way that structural complexity is limited ? Examples include avoiding cyclic dependencies, reducing dependencies, and using techniques such as dependency injection.				
	Does the system include few or no (i.e., limited) sources of nondeterminism ? This helps to limit the behavioral complexity that comes with unconstrained parallelism, which in turn simplifies testing.				

A.10 Usability

Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Supporting user initiative	Does the system support operation canceling ? Does the system support operation undoing ?				
	Does the system support operations to be paused and later resumed ? Examples are pausing the download of a file in a web browser and allowing the user to retry an incomplete (and failed) download.				
	Does the system support operations to be applied to groups of objects (aggregation)? For example, does it allow you to see the cumulative size of a number of files that are selected in a file browser window?				
Support system initiative	Does the system provide assistance to the user based on the tasks that he or she is performing (by maintaining a task model)? Examples include: <ul style="list-style-type: none"> ▪ Validation of input data ▪ Drawing user 				

- | | | | | |
|--|---|--|--|--|
| | attention to changes
in the UI | | | |
| | <ul style="list-style-type: none">▪ Maintaining UI
consistency▪ Adding toolbars and
menus to help users
find functionality | | | |

	<p>provided by the UI</p> <ul style="list-style-type: none"> ▪ Using wizards or other techniques to guide users in performing key user scenarios 			
	<p>Does the system support adjustments to the UI with respect to the class of users (by maintaining a user model)? Examples include supporting UI customization (including localization) and supporting accessibility.</p>			
	<p>Does the system provide appropriate feedback to the user based on the system characteristics (by maintaining a system model)?</p> <p>Examples include:</p> <ul style="list-style-type: none"> ▪ Avoiding blocking the user while handling long-running requests ▪ Providing feedback on action progress (i.e., progress bars) ▪ Displaying user-friendly errors without exposing sensitive data by managing exceptions ▪ Adjusting the UI with respect to screen size and resolution 			

A.11 Further Reading

The tactics catalog from which the questionnaires are derived can be found in L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice* (4th ed.), Addison-Wesley, 2021.

An analysis of quality attribute data from SEI ATAMs, showing which qualities are the most common in practice, can be found in I. Ozkaya, L. Bass, R. Sangwan, and R. Nord, “Making Practical Use of Quality Attribute Information”, *IEEE Software*, March/April 2008, and in a later study by S. Bellomo, I. Gorton, and R. Kazman, “Insights from 15 Years of ATAM Data: Towards Agile Architecture”, *IEEE Software*, 32:5, 38-45, September/October 2015.

Author Bio

Humberto Cervantes is a professor at Universidad Autónoma Metropolitana Iztapalapa in Mexico City. His primary research interest is software architecture design process tool development methods. He holds the Software Architecture Professional and ATAM Evaluator certificates from the SEI.

Rick Kazman is the Danny and Elsa Lui Distinguished Professor of Information Technology Management at the University of Hawaii. He was involved in creating the ATAM (Architecture Tradeoff Analysis Method) and the Titan and DV8 tools for architecture analysis.