The Addison-Wesley Signature Series

A Vaughn Vernon Signature Book

Continuous Architecture in Practice

Software Architecture in the Age of Agility and DevOps

Murat Erder
Pierre Pureur
Eoin Woods

Foreword by Kurt Bittner

Continuous Architecture Principles

Principle 1: Architect products; evolve from projects to products.

Principle 2: Focus on quality attributes, not on functional requirements.

Principle 3: Delay design decisions until they are absolutely necessary.

Principle 4: Architect for change—leverage the "power of small."

Principle 5: Architect for build, test, deploy, and operate.

Principle 6: Model the organization of your teams after the design of the system you are working on.

Praise for *Continuous Architecture in Practice*

"I am continuously delighted and inspired by the work of these authors. Their first book laid the groundwork for understanding how to evolve the architecture of a software-intensive system, and this latest one builds on it in some wonderfully actionable ways."

—Grady Booch, Chief Scientist for Software Engineering, IBM Research

"*Continuous Architecture in Practice* captures the key concerns of software architects today, including security, scalability and resilience, and provides valuable insights into managing emerging technologies such as machine/deep learning and blockchain. A recommended read!"

—Jan Bosch, Professor of Software Engineering and Director of the Software Center at Chalmers University of Technology, Sweden

"*Continuous Architecture in Practice* is a great introduction to modern-day software architecture, explaining the importance of shifting architectural thinking 'left' in order to form a set of firm foundations for delivery and continuous architecture evolution. I really liked the coverage of quality attributes, with a real-world case study providing a way to highlight the real-world complexities of the trade-offs associated with different solutions. The set of references to other material is impressive too, making this book incredibly useful for readers new to the domain of software architecture."

—Simon Brown, author of Software Architecture for Developers

"Focus on software architecture can get lost when talking about agile software practices. However, the importance of architecture in software systems has always been and continues to be relevant. The authors address this important topic with their second book on Continuous Architecture. This time they provide advice on aspects that will make or break your system, from data to security, scalability and resilience. A much recommended book that offers practical guidance for anyone developing systems in today's rapidly evolving technology landscape."

—Ivar Jacobson

"This book continues the journey where its predecessor left off. Software today is never-ending, and true to its name, this book looks at continuing trends and applies Continuous Architecture principles using practical examples. The authors avoid the trap of picking popular tools whose relevance quickly expire, choosing instead to look at those trends that should influence and shape architecture decisions. This book will be essential reading for any person wanting to design and architect software systems that continue to keep up with the times."

—Patrick Kua, CTO Coach and Mentor

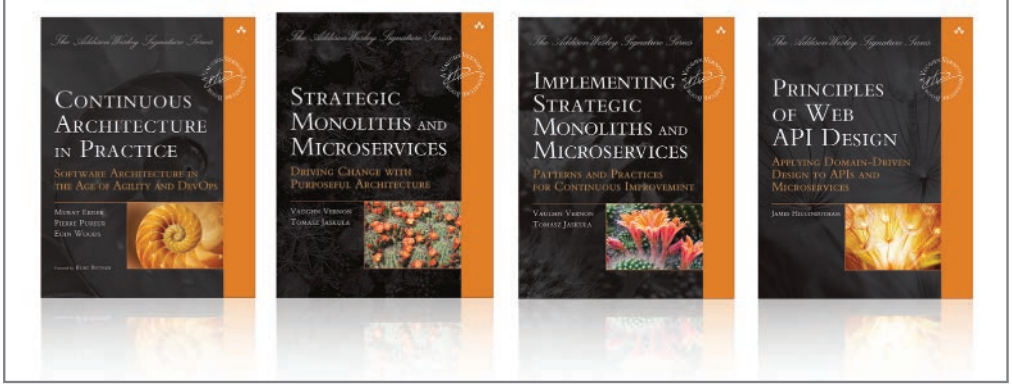
"In the two-decade-old conflict between 'big upfront design' and 'emergent architecture,' software architects have often had a hard time finding a meaningful compromise. In *Continuous Architecture in Practice*, Erder, Pureur, and Woods provide them with a proven path. This book is a big leap forward: I liked the more systematic use of architectural tactics—a design artifact that has not been exploited as much as it should. And that they brought the concept of architectural technical debt to a more prominent position in the process of making technical and managerial decisions."

—Philippe Kruchten, software architect

"It's high time that Agile architecture evolved from oxymoron to what it really needs to be, a lean, enabling practice that accelerates development and delivery of the next generation of resilient and scalable enterprise class systems. *Continuous Architecture in Practice* is another quantum step toward that goal and provides practical guidance toward creating designs that are responsive to changing requirements and technologies."

—Dean Leffingwell, creator of SAFe

Pearson Addison-Wesley Signature Series



Visit informit.com/awss/vernon for a complete list of available publications.

The **Pearson Addison-Wesley Signature Series** provides readers with practical and authoritative information on the latest trends in modern technology for computer professionals. The series is based on one simple premise: great books come from great authors.

Vaughn Vernon is a champion of simplifying software architecture and development, with an emphasis on reactive methods. He has a unique ability to teach and lead with Domain-Driven Design using lightweight tools to unveil unimagined value. He helps organizations achieve competitive advantages using enduring tools such as architectures, patterns, and approaches, and through partnerships between business stakeholders and software developers.

Vaughn's Signature Series guides readers toward advances in software development maturity and greater success with business-centric practices. The series emphasizes organic refinement with a variety of approaches—reactive, object, and functional architecture and programming; domain modeling; right-sized services; patterns; and APIs—and covers best uses of the associated underlying technologies.

Continuous Architecture in Practice

Software Architecture in the Age of Agility and DevOps

Murat Erder
Pierre Pureur
Eoin Woods

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2021933714

Cover image: KMNPhoto / Shutterstock

Page 12, Figure 1.2: welder, Factory_Easy/Shutterstock; bridge, Z-art/Shutterstock; architects, Indypendenz/Shutterstock; Hotel Del Coronado, shippee/Shutterstock.

Page 18, Figure 1.5: toolbox, WilleeCole Photography/Shutterstock; checkmark, Alev Bagater/Shutterstock; wrench, Bonezboyz/Shutterstock.

Page 38, Figure 2.6: Kruchten, P., R. Nord & I. Ozkaya. Managing Technical Debt: Reducing Friction in Software Development, 1st Ed., 2019. Reprinted by permission of Pearson Education, Inc.

Page 145, Figure 5.7: cargo ship, Faraways/Shutterstock.

ISBN-13: 978-0-13-652356-7
ISBN-10: 0-13-652356-0

ScoutAutomatedPrintCode

To Hakan, Ozan, and Pinar
—M.E.

To Kathy
—P.P.

To my family
—E.W.

This page intentionally left blank

Contents

Series Editor Foreword

My Signature Series is designed and curated to guide readers toward advances in software development maturity and greater success with business-centric practices. The series emphasizes organic refinement with a variety of approaches—reactive, object, as well as functional architecture and programming; domain modeling; right-sized services; patterns; and APIs—and covers best uses of the associated underlying technologies.

From here I am focusing now on only two words: organic refinement.

The first word, *organic*, stood out to me recently when a friend and colleague used it to describe software architecture. I have heard and used the word *organic* in connection with software development, but I didn't think about that word as carefully as I did then when I personally consumed the two used together: *organic architecture*.

Think about the word *organic*, and even the word *organism*. For the most part these are used when referring to living things, but are also used to describe inanimate things that feature some characteristics that resemble life forms. *Organic* originates in Greek. Its etymology is with reference to a functioning organ of the body. If you read the etymology of *organ*, it has a broader use, and in fact *organic* followed suit: body organs; to implement; describes a tool for making or doing; a musical instrument.

We can readily think of numerous organic objects—living organisms—from the very large to the microscopic single-celled life forms. With the second use of organism, though, examples may not as readily pop into our mind. One example is an organization, which includes the prefix of both *organic* and *organism*. In this use of *organism*, I'm describing something that is structured with bidirectional dependencies. An organization is an organism because it has organized parts. This kind of organism cannot survive without the parts, and the parts cannot survive without the organism.

Taking that perspective, we can continue applying this thinking to nonliving things because they exhibit characteristics of living organisms. Consider the atom. Every single atom is a system unto itself, and all living things are composed of atoms. Yet, atoms are inorganic and do not reproduce. Even so, it's not difficult to think of atoms as living things in the sense that they are endlessly moving, functioning. Atoms even bond with other atoms. When this occurs, each atom is not only a single system

unto itself, but becomes a subsystem along with other atoms as subsystems, with their combined behaviors yielding a greater whole system.

So then, all kinds of concepts regarding software are quite organic in that nonliving things are still "characterized" by aspects of living organisms. When we discuss software model concepts using concrete scenarios, or draw an architecture diagram, or write a unit test and its corresponding domain model unit, software starts to come alive. It isn't static, because we continue to discuss how to make it better, subjecting it to refinement, where one scenario leads to another, and that has an impact on the architecture and the domain model. As we continue to iterate, the increasing value in refinements leads to incremental growth of the organism. As time progresses so does the software. We wrangle with and tackle complexity through useful abstractions, and the software grows and changes shapes, all with the explicit purpose of making work better for real living organisms at global scales.

Sadly, software organics tend to grow poorly more often than they grow well. Even if they start out life in good health they tend to get diseases, become deformed, grow unnatural appendages, atrophy, and deteriorate. Worse still is that these symptoms are caused by efforts to refine the software, that go wrong instead of making things better. The worst part is that with every failed refinement, everything that goes wrong with these complexly ill bodies doesn't cause their death. Oh, if they could just die! Instead, we have to kill them and killing them requires nerves, skills, and the intestinal fortitude of a dragon slayer. No, not one, but dozens of vigorous dragon slayers. Actually, make that dozens of dragon slayers who have really big brains.

That's where this series comes into play. I am curating a series designed to help you mature and reach greater success with a variety of approaches—reactive, object, and functional architecture and programming; domain modeling; right-sized services; patterns; and APIs. And along with that, the series covers best uses of the associated underlying technologies. It's not accomplished at one fell swoop. It requires organic refinement with purpose and skill. I and the other authors are here to help. To that end, we've delivered our very best to achieve our goal.

That's why I and other authors in this series have chosen this book to be among our own. We know value when we see it. Here's what we thought of *Continuous Architecture in Practice*.

We sensed the power of *Continuous Architecture in Practice*. Do you? An architecture that is continuous is such because it possesses distinct organic properties. Having characteristics of living organisms, it changes with its circumstances and stands as a sound foundation and protection from numerous negative influences. Such architectures are continually improved by organic refinement because they are driven by, and rapidly respond to, new and changing nonfunctional requirements that support the ongoing call for innovative functional requirements. Further, this book provides instruments for organization and tooling that will help make

architectures provide structure and evolve with the growing software. The *in practice* part calls for learning from the practices of experienced and mature architects, and leveraging them in one's own work.

If followed as the authors have intended, the wisdom embodied in *Continuous Architecture in Practice* will help you make wise, practical decisions, not merely intellectual ones. The wisdom between the covers has been provided by software professionals with several decades of hard-won experience working in large and complex enterprises. Murat Erder, Pierre Pureur, and Eoin Woods have likewise delivered their very best, an end-to-end set of decision-making tools, patterns, and advice. You will not regret learning from them.

—Vaughn Vernon, series editor

This page intentionally left blank

Foreword

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

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

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

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

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

Happy architecting!

—Kurt Bittner

This page intentionally left blank

Introduction

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

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

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

Continuous Architecture in Practice is organized as follows:

- In Chapter 1, we provide context, define terms, and provide an overview of the case study that will be used throughout each chapter (more details for the case study are included in Appendix A).
- In Chapter 2, our key ideas are laid out, providing the reader with an understanding of how to perform architectural work in today's software development environment.

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

- In Chapters 3 through 7, we explore a number of architecture topics that are central to developing modern applications: data, security, scalability, performance, and resilience. We explain how software architecture, in particular the Continuous Architecture approach, can help to address each of those architectural concerns while maintaining an agile way of working that aims to continually deliver change to production.
- In Chapters 8 and 9, we look at what is ahead. We discuss the role of architecture in dealing with emerging technologies and conclude with the challenges of practicing architecture today in the era of agile and DevOps as well as potential ways to meet those challenges.

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

To keep the scope of this book manageable and focused on what has changed since our last book, we assume that readers are familiar with the basics of mainstream technical topics such as information security, cloud computing, microservice-based architecture, and common automation techniques such as automated testing and deployment pipelines. We expect that our readers are also familiar with the fundamental techniques of architectural design, how to create a visual model of their software, and associated techniques such as the domain-driven design (DDD) approach.[4] For those who feel unsure about architectural design fundamentals, we suggest starting with a well-defined approach such as the Software Engineering Institute's attribute-driven design[5] or a simpler approach such as the one outlined in chapter 7 of *Software Systems Architecture*. Software modeling, although neglected for a few years, seems to be returning to mainstream practice. For those who missed it first time around, chapter 12 of

2. Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice* (Addison-Wesley, 2012).
3. Nick Rozanski and Eoin Woods, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives* (Addison-Wesley, 2012).
4. For more information on DDD, please see Vaughn Vernon, Implementing Domain-Driven Design (Addison-Wesley, 2013).
5. Humberto Cervantes and Rick Kazman, *Designing Software Architectures: A Practical Approach* (Addison-Wesley, 2016). The AAD approach is also outlined in Bass, Clements, and Kazman, *Software Architecture in Practice*, chapter 17.

Software Systems Architecture provides a starting point, and Simon Brown's books[6,7] are a more recent and very accessible introduction to it.

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

We also assume an existing knowledge of agile development and so do not provide in-depth discussions of software development life cycle processes such as agile, Scrum, and the Scaled Agile Framework (SAFe), nor do we discuss software deployment and operation approaches, such as DevSecOps, in any depth. We deliberately do not include details on any specific technology domain (e.g., database, security, automation). We of course refer to these topics where relevant, but we assume our readers are generally familiar with them. We covered these topics, except for technology details, in *Continuous Architecture*. Also, please note that terms defined in the glossary are highlighted in **bold** the first time they appear in this book.

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

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

No matter what the perception, we are confident that architects who exhibit the notorious qualities of abstract mad scientists, technology tinkerers, or presentation

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

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

junkies are a minority of practitioners. A majority of architects work effectively as part of software delivery teams, most of the time probably not even calling themselves architects. In reality, all software has an architecture (whether or not it is well understood), and most software products have a small set of senior developers who create a workable architecture whether or not they document it. So perhaps it is better to consider architecture to be a skill rather than a role.

We believe that the pendulum has permanently swung away from historical, document-centric software architectural practices and perhaps from conventional enterprise architecture as well. However, based on our collective experience, we believe that there is still a need for an architectural approach that can encompass agile, continuous delivery, DevSecOps, and cloud-centric computing, providing a broad architectural perspective to unite and integrate these approaches to deliver against our business priorities. The main topic of this book is to explain such an approach, which we call Continuous Architecture, and show how to effectively utilize this approach in practice.

Acknowledgments

Soon after the first edition of Continuous Architecture was published, we started getting feedback from our readers along the lines of *great book, but I wish you had provided more practical advice on how to implement Continuous Architecture principles and tools. What happened to Data? That topic seems to be missing from your book*. In the spirit of Continuous Architecture, we immediately started applying this feedback to the book, and planned for a revised edition.

However, it soon became apparent to us that we needed more than a new edition of *Continuous Architecture* to fully address all that feedback. What was needed was a new book, one that explains how Continuous Architecture may help with the practical, day-to-day challenges that development teams face in today's complex environment of rapidly changing technologies, growing masses of data, and stringent quality attribute requirements associated with modern systems, such as security, scalability, performance, and resiliency. As you can guess, this project took longer than initially anticipated due to its increased scope and the unforeseen challenges of writing a manuscript in the middle of a pandemic. However, we have managed to bring this project past the finish line and would like to thank the following people for encouraging us and helping us in this endeavor.

Kurt Bittner and John Klein for taking the time to review our manuscript and providing invaluable feedback and guidance. Grady Booch, Jan Bosch, Simon Brown, Ivar Jacobson, Patrick Kua, Philippe Kruchten, and Dean Leffingwell for their feedback and endorsements, and Peter Eeles, George Fairbanks, Eltjo Poort, and Nick Rozanski for many thought-provoking discussions over the years.

Richard Gendal Brown, CTO of R3, for providing us with invaluable advice on distributed ledger technologies and blockchain. Andrew Graham for sense-checking our trade finance case study. Mark Stokell for providing content for data science-related further reading. Philippe Kruchten, Robert Nord, and Ipek Ozkaya for permission to reference content and diagrams from their book, *Managing Technical Debt*. And, of course, all of our *Continuous Architecture* readers for providing the feedback that led us to create this book.

Finally, we would like to thank Haze Humbert, Julie Nahil, and Menka Mehta from Pearson, who supported and collaborated with us during the entire process; and Vaughn Vernon, who thought that this book would be a valuable addition to his Addison-Wesley Signature Series.

This page intentionally left blank

About the Authors

Murat Erder has more than twenty-five years' experience in the software industry working for software vendors, management consultancies and large international banks. During his career Murat has had a variety of roles, from developer, to software architect, to management consultant. Murat's corporate IT roles cover the areas of data, integration, architecture and working as a CTO. He is co-author of the book *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World* (2015) and has presented on this topic at a range of conferences, including SEI Saturn, O'Reilly Software Architecture and GOTOLondon.

Pierre Pureur is an experienced software architect, with extensive innovation and application development background, vast exposure to the financial services industry, broad consulting experience and comprehensive technology infrastructure knowledge. His past roles include serving as Chief Enterprise Architect for a major financial services company, leading large architecture teams, managing large-scale concurrent application development projects and directing innovation initiatives, as well as developing strategies and business plans. He is coauthor of the book *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World* (2015) and has published many articles and presented at multiple software architecture conferences on this topic.

Eoin Woods is the Chief Technology Officer of Endava, where he guides technical strategy, leads capability development and directs investment in emerging technologies. Prior to joining Endava, Eoin worked in the software engineering industry for twenty years, developing system software products and complex applications in the capital markets domain. His main technical interests are software architecture, DevOps and software security and resilience. He is coauthor of the book *Software Systems Architecture*, is a frequent speaker at industry events and was the recipient of the 2018 Linda M. Northrup Award for Software Architecture, awarded by the SEI at Carnegie Mellon University.

This page intentionally left blank

Chapter 2

Architecture in Practice: Essential Activities

The architect should strive continually to simplify.

—Frank Lloyd Wright

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

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

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

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

A key aspect about the topic of architecture and architects is that it traditionally assumes one all-seeing and wise individual is doing architecture. In Continuous Architecture, we propose to move away from this model. We refer to "architecture work" and "architectural responsibility" instead. These terms point to the

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

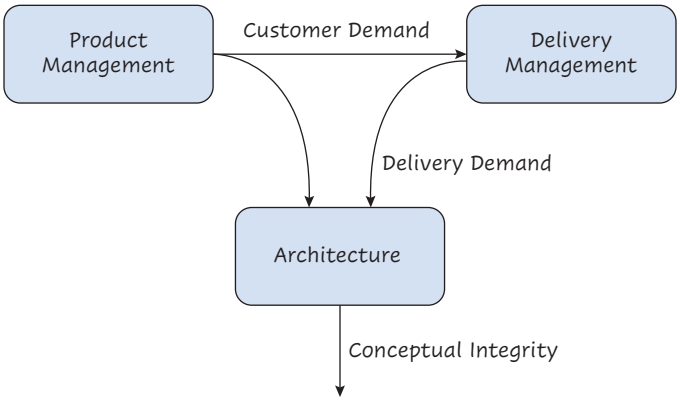


Figure 2.1 *Balancing role of architecture*

importance of the activities, while emphasizing the responsibility of the team rather than of a single person.

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

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

Essential Activities Overview

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

- *Focus on quality attributes*, which represent the key cross-cutting requirements that a good architecture should address. Quality attributes—performance,

2. Frederick P. Brooks Jr., *The Mythical Man-Month: Essays on Software Engineering* (Addison-Wesley, 1995).

scalability, and security, among others—are important because they drive the most significant architectural decisions that can make or break a software system. In subsequent chapters, we discuss in detail architectural tactics that help us address quality attributes.

- *Drive architectural decisions*, which are the primary unit of work of architectural activities. Continuous Architecture recommends explicitly focusing on architectural decisions because if we do not understand and capture architectural decisions, we lose the knowledge of tradeoffs made in a particular context. Without this knowledge, the team is inhibited from being able to support the long-term evolution of the software product. As we refer to our case study, we highlight key architectural decisions the team has made.
- *Know your technical debt*, the understanding and management of which is key for a sustainable architecture. Lack of awareness of technical debt will eventually result in a software product that cannot respond to new **feature** demands in a cost-effective manner. Instead, most of the team's effort will be spent on working around the technical debt challenges—that is, paying back the debt.
- *Implement feedback loops*, which enable us to iterate through the software development life cycle and understand the impact of architectural decisions. Feedback loops are required so that the team can react quickly to developments in requirements and any unforeseen impact of architectural decisions. In today's rapid development cycles, we need to be able to course-correct as quickly as possible. Automation is a key aspect of effective feedback loops.

Figure 2.2 depicts the Continuous Architecture loop that combines these elements.

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

- Architectural decisions directly impact the production environment.
- Feedback loops measure the impact of architectural decisions and how the software system is fulfilling quality attribute requirements.

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

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

- Quality attribute requirements and technical debt help us prioritize architectural decisions.
- Architectural decisions can add or remove technical debt.

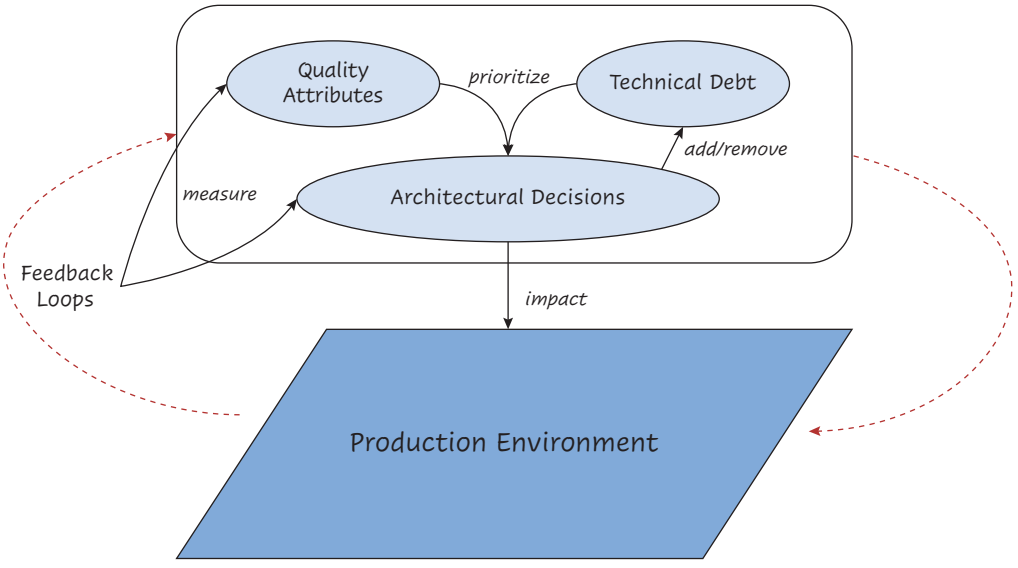


Figure 2.2 *Essential activities of architecture*

It might come as a surprise that we do not talk about models, perspectives, views, and other architecture artifacts. These are incredibly valuable tools that can be leveraged to describe and communicate the architecture. However, if you do not undertake the essential activities we emphasize, architecture artifacts on their own will be insufficient. In other words, models, perspectives, views, and other architecture artifacts should be considered as a means to an end—which is to create a sustainable software system.

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

Architectural Decisions

If you ask software practitioners what the most visible output is from architectural activities, many will likely point to a fancy diagram that highlights the key components and their interactions. Usually, the more color and complexity, the better.

The diagram is typically too difficult to read on a normal page and requires a special large-scale printer to produce. Architects want to look smart, and producing a complex diagram shows that the architect can solve extremely difficult problems! Though such diagrams give the authors and readers the false sense of being in control, they normally have limited impact on driving any architectural change. In general, these diagrams are rarely understood in a consistent manner and provide limited insight without a voiceover from the diagram's author. In addition, diagrams are hard to change, which ends up in a divergence from the code running in the production environment that adds confusion when making architectural decisions.

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

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

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

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

5. https://adr.github.io

6. Erder and Pureur, "Evolving the Architecture," in *Continuous Architecture*, 63–101.

Making and Governing Architectural Decisions

Let us look at the different types of architectural decisions in an enterprise. Figure 2.3 demonstrates our recommended approach to making architectural decisions in a typical enterprise.[7]

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

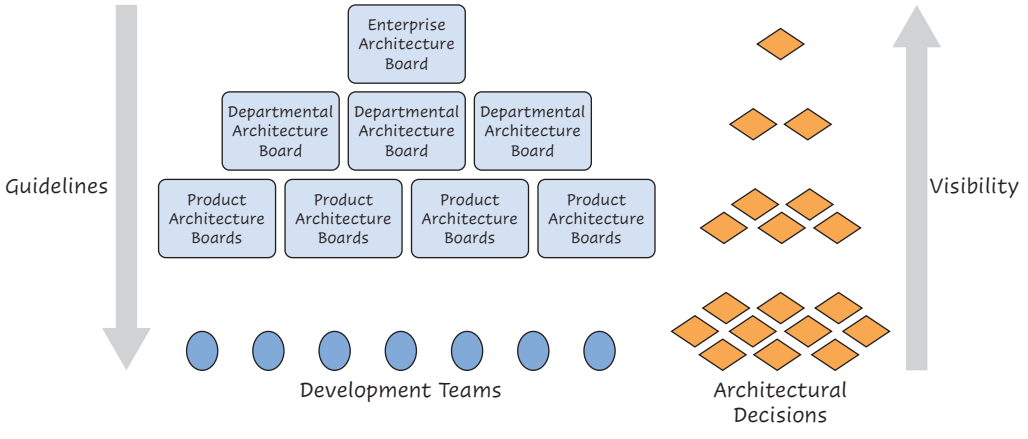


Figure 2.3 *Levels of architectural decisions*

- *Guidelines:* In reality, the probability of development teams compromising the architecture is greatly reduced if they are given clear guidelines to adhere to.

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

For example, if there are clear guidelines around where and how to implement stored procedures, then the risk of creating a brittle architecture by writing stored procedures in random parts of the architecture can be avoided.[8] If you go back to Figure 2.3, you see that the main job of higher governance bodies is not to make decisions but to define guidelines. The recommended approach is that there should be fewer principles the higher you go in the organization.

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

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

- Applying principle 4, *Architect for change—leverage the "power of small,"* results in loosely coupled cohesive components. The architectural decisions within a component will have limited impact on other components. Some architectural decisions will still cut across components (e.g., minimally how to define the components and their integration patterns), but these decisions can also be addressed independently of component-specific decisions.

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

9. https://dddcommunity.org/learning-ddd/what_is_ddd

- Applying principle 6, *Model the organization of your teams after the design of the system you are working on*, results in collaborative teams that focus on delivering a set of components. This means that the knowledge sharing of relevant architectural decisions is more natural because the team is already operating in a collaborative manner.

Architectural Decisions in Agile Projects

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

By clearly defining all known architectural decisions, we are basically creating an architectural backlog. This list includes the decisions you have made and the ones you know you have to make. Obviously, the list of architectural decisions will evolve as you make decisions and develop your product. What is important is to have a list of known architectural decisions and decide on which ones you need to address immediately. Remember principle 3, *Delay design decisions until they are absolutely necessary*.

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

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

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

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

10. Erder and Pureur, "Continuous Architecture in the Enterprise," in *Continuous Architecture*, 215–254.

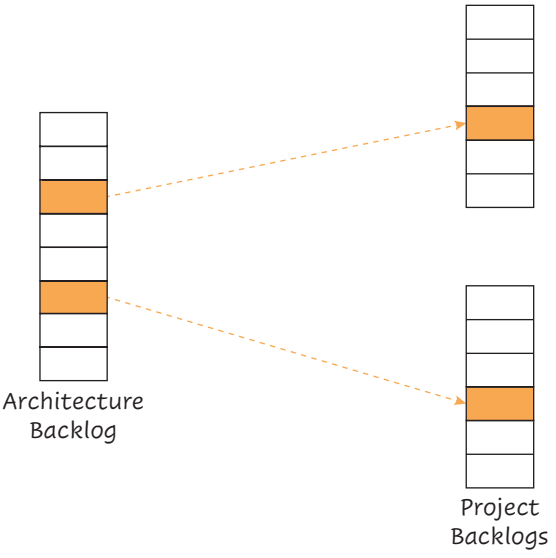


Figure 2.4 *Architectural decision and product backlogs*

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

Table 2.1 *Decision Log Entry Example*

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

Quality Attributes

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

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

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

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

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

11. A more humorous criticism of nonfunctional requirements is the view that the term indicates that the requirement itself is nonfunctional.
12. Nick Rozanski and Eoin Woods, *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives* (Addison-Wesley, 2012).
13. International Organization for Standardization and International Electrotechnical Commission, *ISO/IEC 25010:2011 Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models* (2011). https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

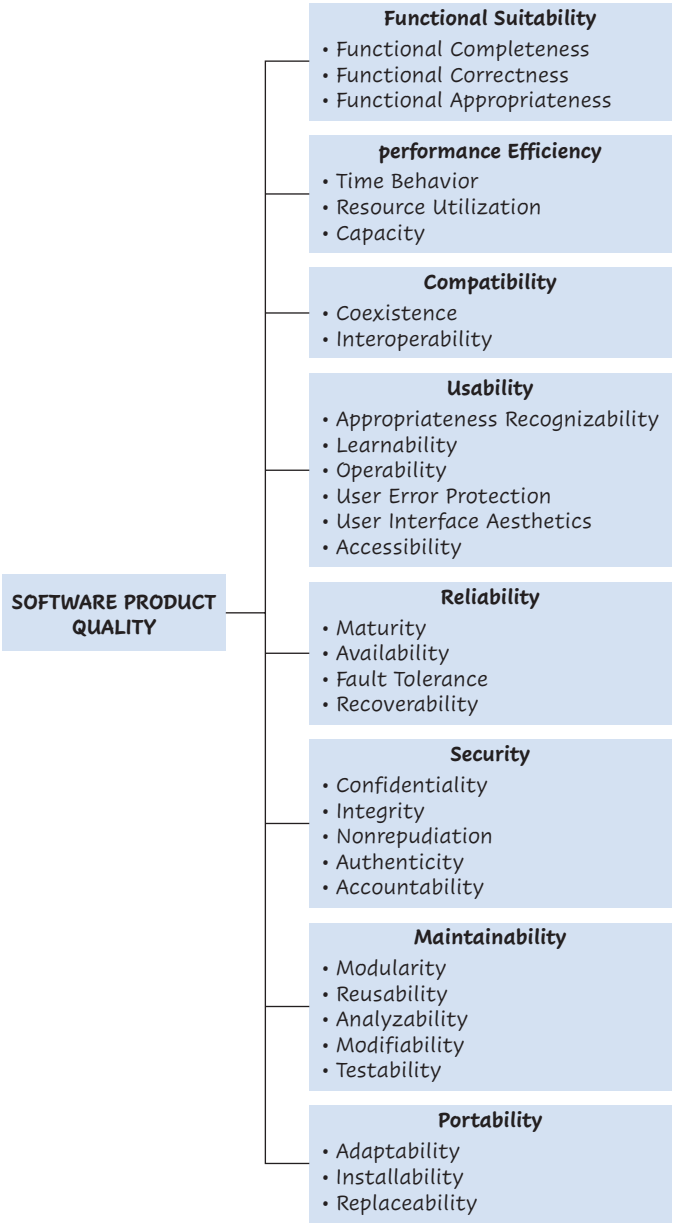


Figure 2.5 *Product quality model*

It is difficult to express quality attributes outside of a particular system context. For example, latency may be nice to have in a tax-filing application but disastrous for an autopilot. This makes it challenging to adopt such frameworks in their entirety,

and defining a complete list of all quality attributes can be seen as an unnecessary academic exercise.

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

Experienced software practitioners know that a given set of functional capabilities can often be implemented by several different architectures with varying quality attribute capabilities. You can say that architectural decisions are about trying to balance tradeoffs to find a good enough solution to meet your functional and quality attribute requirements.

Quality Attributes and Architectural Tactics

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

However, our view is that quality attributes drive the architecture design. As stated by Bass, Clements, and Kazman, "Whether a system will be able to exhibit its desired (or required) quality attributes is substantially determined by its architecture."[14] We need to make architectural decisions to satisfy quality attributes, and those decisions often are compromises, because a decision made to better implement a given quality attribute may have a negative impact on the implementation of other quality attributes. Accurately understanding quality attribute requirements and tradeoffs is one of the most critical prerequisites to adequately architect a system. Architectural decisions are often targeted to find the least-worst option to balance the tradeoffs between competing quality attributes.

Architectural tactics are how we address quality attributes from an architectural perspective. An architectural tactic is a decision that affects the control of one or more quality attribute responses. Tactics are often documented in catalogs in order

14. Bass, Clements, and Kazman, *Software Architecture in Practice*, 26.

to promote reuse of this knowledge among architects. We refer to architectural tactics throughout the book, in particular in chapters 5 through 7, that focus on specific quality attributes.

Working with Quality Attributes

In the Continuous Architecture approach, our recommendation is to elicit and describe the quality attribute requirements that will be used to drive architectural decisions. But how do we describe quality attributes? A quality attribute name by itself does not provide sufficiently specific information. For example, what do we mean by *configurability*? Configurability could refer to a requirement to adapt a system to different infrastructures—or it could refer to a totally different requirement to change the business rules of a system. Attribute names such as "availability," "security," and "usability" can be just as ambiguous. Attempting to document quality attribute requirements using an unstructured approach is not satisfactory, as the vocabulary used to describe the quality attributes may vary a lot depending on the perspective of the author.

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

We recommend leveraging the utility tree technique from the architecture tradeoff analysis method, or **ATAM**.[15] Documenting architecture scenarios that illustrate quality attribute requirements is a key aspect of this technique.

Building the Quality Attributes Utility Tree

We do not go into details of the **ATAM utility tree**, which is covered in our original book.[16] The most important aspect is to clearly understand the following three attributes for each scenario:

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

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

16. Erder and Pureur, "Getting Started with Continuous Architecture: Requirements Management," in *Continuous Architecture*, 39–62.

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

Another attribute you can include in defining the scenario is

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

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

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

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

Technical Debt

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

> Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. . . . The danger occurs when the debt is

> not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise.[17]

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

> In software-intensive systems, technical debt consists of design or implementation constructs that are expedient in the short term but that set up a technical context that can make future change more costly or impossible. Technical debt is a contingent liability whose impact is limited to internal system qualities—primarily, but not only, maintainability and evolvability.[18]

This is a good definition because it focuses more on the impact of technical debt and does not strictly follow the financial debt metaphor—which, though useful, is not a fully accurate way to represent the topic. The focus on maintainability and evolvability is key to how to think about technical debt. It implies that if your system is not expected to evolve, the focus on technical debt should be minimal. For example, software written for the *Voyager* spacecraft should have very limited focus on technical debt[19] because it is not expected to evolve and has limited maintenance opportunities.

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

- *Code:* This category includes expediently written code that is difficult to maintain and evolve (i.e., introduce new features). The Object Management Group (OMG)'s *Automated Technical Debt Measure* specification[20] can be used by source code analysis tools to measure this aspect. You can view the specification as standardized best practices for common topics such as managing loops,

17. Ward Cunningham, "The WyCash Portfolio Management System," *ACM SIGPLAN OOPS Messenger* 4, no. 2 (1992): 29–30.
18. Philippe Kruchten, Rod Nord, and Ipek Ozkaya, *Managing Technical Debt: Reducing Friction in Software Development* (Addison-Wesley, 2019).
19. Or at least no intentional debt. It is almost impossible to avoid creating unintentional debt. See Kruchten, Nord, and Ozkaya, Managing Technical Debt, principle 3, "All systems have technical debt."
20. Object Management Group, *Automated Technical Debt Measure* (December 2017). https://www.omg.org/spec/ATDM

initialization of variables, and so on. Because this book is more about architecture than implementation, we do not discuss this aspect of technical debt any further.

- *Architecture:* Debt in this category is the result of architectural decisions made during the software development process. This type of technical debt is difficult to measure via tools but usually has a more significant impact on the system than other types of debt. For example, the decision to use a database technology that cannot provide the quality attributes required (e.g., using a relational database when a basic key–value database would do) has a significant impact on the scalability and maintainability of a system.
- *Production infrastructure:* This category of technical debt deals with decisions focused on the infrastructure and code that are used to build, test, and deploy a software system. Build-test-deploy is becoming increasingly integral to software development and is the main focus of DevOps. Continuous Architecture sees the build-test-deploy environment as part of the overall architecture, as stated by principle 5, *Architect for build, test, deploy, and operate*.

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

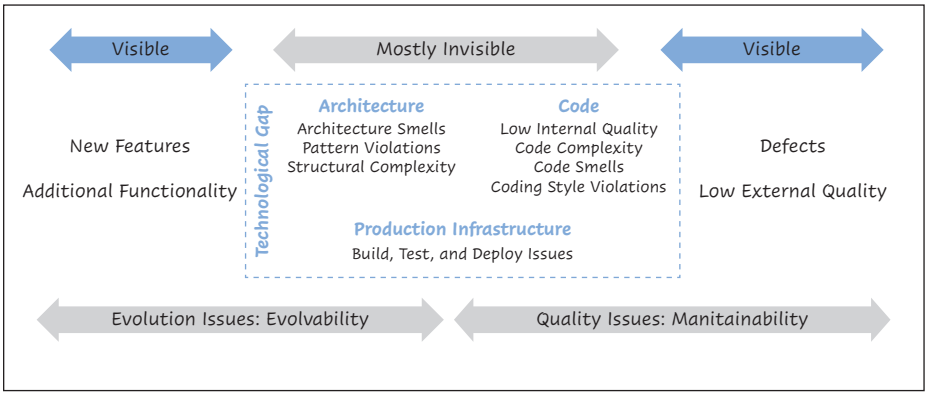


Figure 2.6 *Technical debt landscape. (Source: Kruchten, P., R. Nord & I. Ozkaya, Managing Technical Debt, SEI Series in Software Engineering, Addison-Wesley, 2019.)*

In an interesting article, Alex Yates[21] proposes the term *technical debt singularity*.

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

> Ever accelerating progress of technology and changes in the mode of human life, which gives the appearance of approaching some essential singularity in the history of the race beyond which human affairs, as we know them, could not continue.[22]

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

> So what happens if the interest we owe on our technical debt starts to exceed the number of man-hours in a working day? Well, I call that the technical debt singularity. This is the point at which software development grinds to a halt. If you spend practically all your time firefighting and you can't release often (or ever?) with confidence, I'm afraid I'm talking about you. You've pretty much reached a dead-end with little hope of making significant progress.[23]

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

Capturing Technical Debt

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

21. Alex Yates, "The Technical Debt Singularity," *Observations* (2015). http://workingwithdevs.com/technical-debt-singularity
22. Stanislaw Ulam, "Tribute to John von Neumann," *Bulletin of the American Mathematical Society* 64, no. 3 (1958): 1–49.
23. Yates, "The Technical Debt Singularity."

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

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

Just like the architectural decision backlog, the technical debt registry should be viewable separately. However, it does not need to be managed as a separate item.[24] One effective approach we have observed is to have product backlog items tagged as technical debt. When required, you can easily pull in all technical debt items from the individual project backlogs, as shown in Figure 2.7.

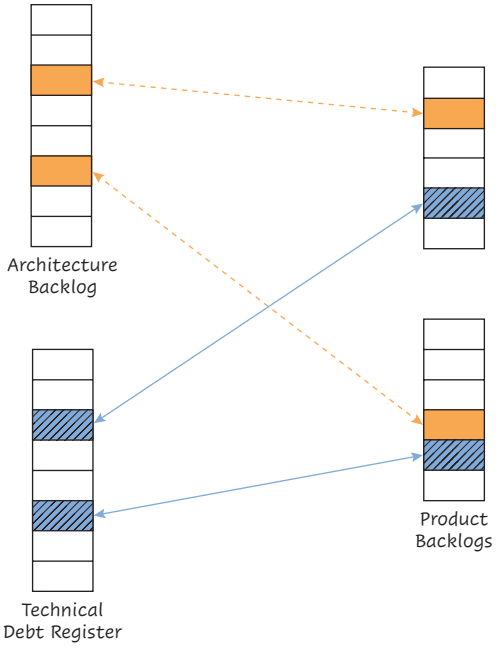


Figure 2.7 *Technical debt registry and backlogs*

24. See Kruchten, Nord, and Ozkaya, *Managing Technical Debt*, chapter 13, for practical advice on this topic.

How to Manage Technical Debt

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

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

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

Now, let us look at how prioritization of backlog items works. In an agile model, it is the product owner who decides what items should be prioritized. Even if you do not operate in a fully agile model, you still have conversations about prioritization and budget with your business stakeholders. If technical debt and its impact is not visible to the business stakeholders, it will always take a back seat to new features. Technical debt items are, by their nature, not clearly visible as features, and they have an impact predominantly on quality attributes.[25] This is where an architectural focus comes in.[26] The objective is to articulate the impact of delaying addressing technical debt items. If we delay addressing technical debt for too long, the software system can hit the technical debt singularity.

Another tactic for making sure technical debt is not lost in the rush for new features is to carve out a proportion of each release to address technical debt. How to categorize your backlog items is a wide area that is not in the scope of this book; however, a compelling view is offered by Mik Kersten.[27] He states that there are four types of items (i.e., flow items) to be considered in the backlog: features, defects, technical debt, and risk (e.g., security, regulatory).

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

---

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

26. See Kruchten, Nord, and Ozkaya, *Managing Technical Debt*, principle 6, "Architecture technical debt has the highest cost of ownership."

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

Feedback Loops: Evolving an Architecture

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

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

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

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

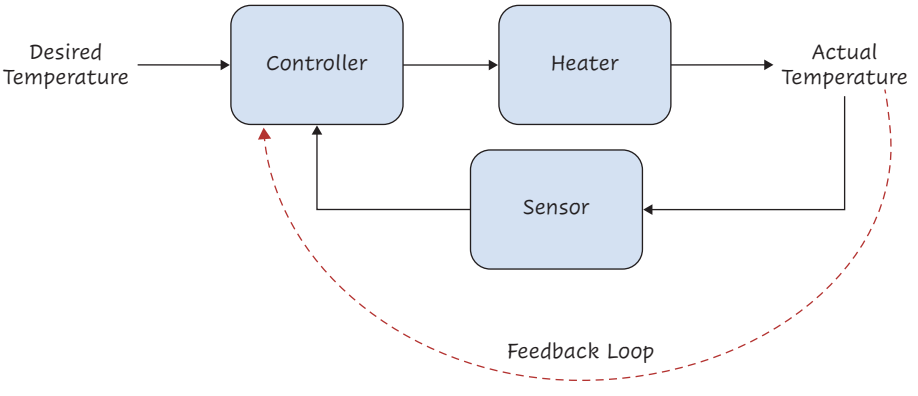


Figure 2.8 *Feedback loop example*

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

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

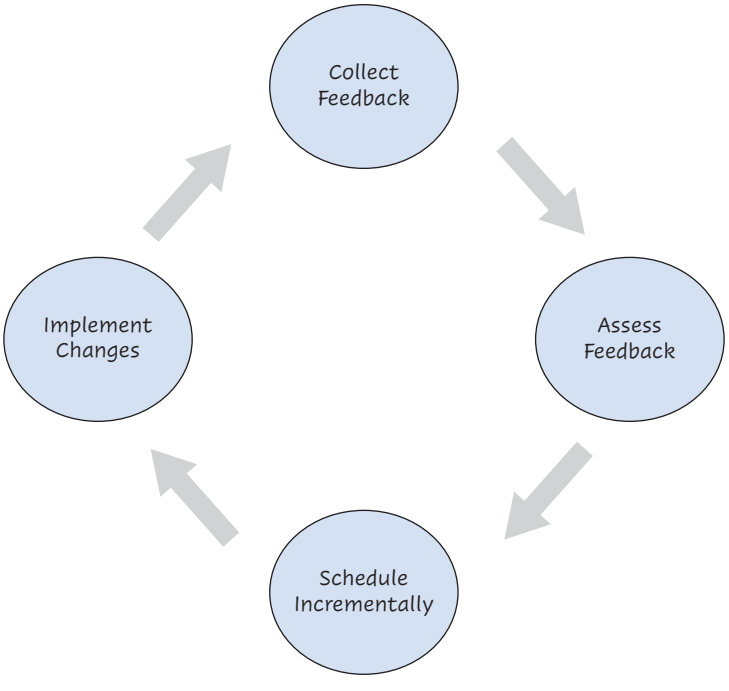


Figure 2.9 *Continuous architecture feedback loop*

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

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

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

- Amount of technical debt being introduced/reduced over time or with each release
- Number of architectural decisions being made and their impact on quality attributes
- Adherence to existing guidelines or standards
- Interface dependencies and coupling between components

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

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

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

Fitness Functions

A key challenge for architects is an effective mechanism to provide feedback loops into the development process of how the architecture is evolving to address quality attributes. In *Building Evolutionary Architectures*,[28] Ford, Parsons, and Kua introduced the concept of the fitness function to address this challenge. They define fitness functions as "an architectural fitness function provides an objective integrity assessment of some architectural characteristics"—where architectural characteristics are what we have defined as quality attributes of a system. These are like the architecturally significant quality attribute scenarios discussed earlier in the chapter.

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

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

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

Continuous Testing

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

28. Neal Ford, Rebecca Parsons, and Patrick Kau, *Building Evolutionary Architectures* (O'Reilly Media, 2017), 15.

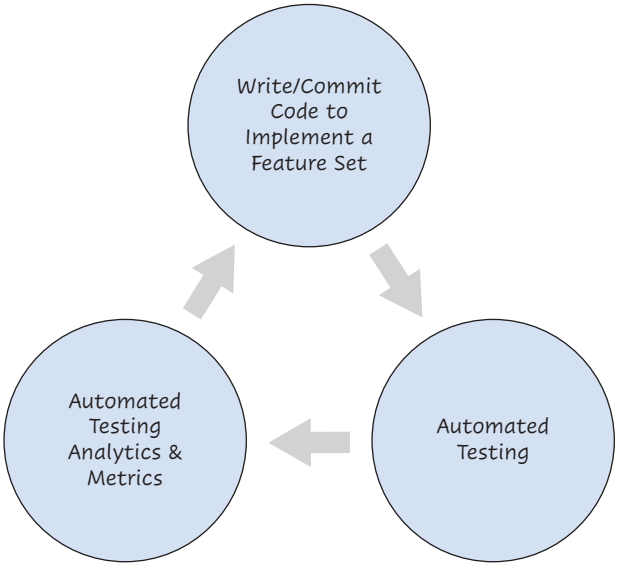


Figure 2.10 *Sample automated testing process*

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

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

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

Continuous testing relies on extensive automation of the testing and deployment processes and on ensuring that every component of the software system can be tested as soon as it is developed. For example, the following tactics[29] can be used by the TFX team for continuous performance testing:

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

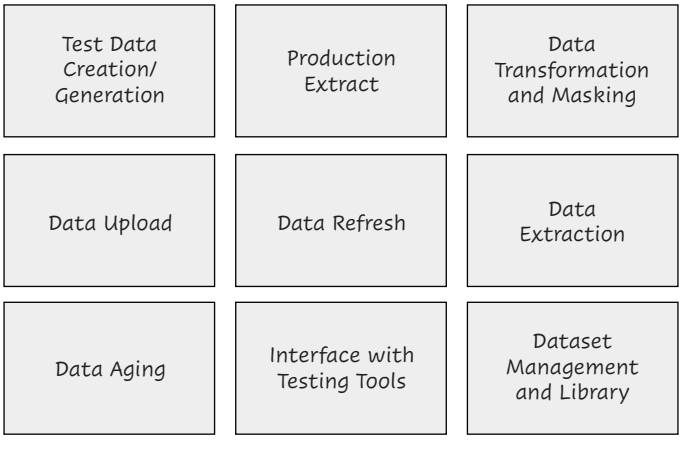


Figure 2.11 *Test data management capabilities*

29. For additional details on those tactics, see Erder and Pureur, "Continuous Architecture and Continuous Delivery," in *Continuous Architecture*, 103–129.

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

Common Themes in Today's Software Architecture Practice

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

Principles as Architecture Guidelines

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

> A declarative statement made with the intention of guiding architectural design decisions in order to achieve one or more qualities of a system.[30]

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

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

30. Eoin Woods, "Harnessing the Power of Architectural Design Principles," *IEEE Software* 33, no. 4 (2016): 15–17.

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

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

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

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

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

Team-Owned Architecture

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

In that context, architecture activities become a team responsibility. Architecture is increasingly becoming a discipline (or skill) rather than a role. We can highlight the key skills required for conducting architectural activities as follows:[31]

31. Eoin Woods, "Return of the Pragmatic Architect," IEEE Software 31, no. 3 (2014): 10–13. https://doi.org/10.1109/MS.2014.69

- *Ability to design*. Architecture is a design-oriented activity. An architect might design something quite concrete, such as a network, or something less tangible, such as a process, but design is core to the activity.
- *Leadership*. Architects are not just technical experts in their areas of specialization: they're technical leaders who shape and direct the technical work in their spheres of influence.
- *Stakeholder focus*. Architecture is inherently about serving a wide constituency of stakeholders, balancing their needs, communicating clearly, clarifying poorly defined problems, and identifying risks and opportunities.
- *Ability to conceptualize and address systemwide concerns*. Architects are concerned about an entire system (or system of systems), not just one part of it, so they tend to focus on systemic qualities rather than on detailed functions.
- *Life cycle involvement*. An architect might be involved in all phases of a system's life cycle, not just building it. Architectural involvement often spans a system's entire life cycle, from establishing the need for the system to its eventual decommissioning and replacement.
- *Ability to balance concerns*. Finally, across all these aspects of the job, there is rarely one right answer in architecture work.

Although we state that architecture is becoming more of a skill than a role, it is still good to have a definition of the role. As mentioned earlier, in *The Mythical Man-Month*,[32] Brooks talks about the conceptual integrity of a software product. This is a good place to start for defining the role of architects—basically, they are accountable for the conceptual integrity of the entity that is being architected or designed.

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

In our first book, *Continuous Architecture*,[33] we provide a detailed overview of the personality traits, skills, and communication mechanisms required for the role of an architect (or to be able to do architectural work).

32. Brooks, *The Mythical Man-Month*.

33. Erder and Pureur, "Role of the Architect," in *Continuous Architecture*, 187–213.

Models and Notations

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

One of the most successful attempts at creating a common notation in the software industry was the Unified Modeling Language (UML), which became an OMG standard in 1997.[34] In the late 1990s and 2000s, it felt as though UML was going to become the default standard for visualizing software. However, it has been waning in popularity in recent years. We are not exactly sure why this is, but one factor is that software engineering is a rapidly expanding and very young profession. As a result, most formalisms are overpowered by new technologies, fads, and ways of working. You can say that the only relevant artifact for developers is code. Any other representation requires extra effort to maintain and therefore becomes quickly outdated as a development team evolves the system.

Another attempt at creating a visual language for software is ArchiMate, which was originally developed in Netherlands and became an Open Group standard in 2008.[35] Unlike UML, which is system focused, ArchiMate attempts to model enterprise architecture artifacts.

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

A more recent attempt at addressing this gap is the C4 model that was created by Simon Brown.[36] This is an interesting approach that addresses some of the challenges with more formal notations. As a philosophy, it tries to create an approach whereby the representation of the architecture is close to the code and can be used by developers.

From a Continuous Architecture perspective, we can make the following observations. As stated at the beginning of the chapter, the core elements required to drive a sustainable architecture are focus on quality attributes, architectural decisions, technical debt, and feedback loops. However, effective communication is critical: *We cannot overcommunicate!* As a result, utilizing a common language to define and

34. https://www.omg.org/spec/UML/About-UML

35. https://pubs.opengroup.org/architecture/archimate3-doc

36. https://c4model.com

communicate architectural artifacts just makes common sense. That the industry has still not found its way does not mean you should not strive for this in your area, be it a system, division, or enterprise.

Although we do not recommend a certain notation, that does not mean graphical communication and effective modeling are unimportant. Following are a few key characteristics that should be considered in determining your approach:[37]

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

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

Patterns and Styles

In 1994, the Gang of Four—Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides—published their seminal book, *Design Patterns*.[38] In this book, they identified 23 patterns that address well-known challenges in object-oriented software development. Almost as important as the solutions they provided is that they introduced the concept of the design pattern and defined a manner to explain the patterns consistently.

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

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

37. https://www.edwardtufte.com

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

Architecture as a Flow of Decisions

As mentioned, the key unit of work of architecture is an architectural decision. The topic of architectural decisions collectively defining the architecture has been prevalent in the industry for some time[39] and is becoming even more prominent in today's world. If you use a Kanban board to combine the view of architectural decisions as a unit of work with common software development practices focused on managing tasks, you can easily say that architecture is just a flow of decisions. Figure 2.12 depicts a simple Kanban board setup that can be used to track the architectural decisions.

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

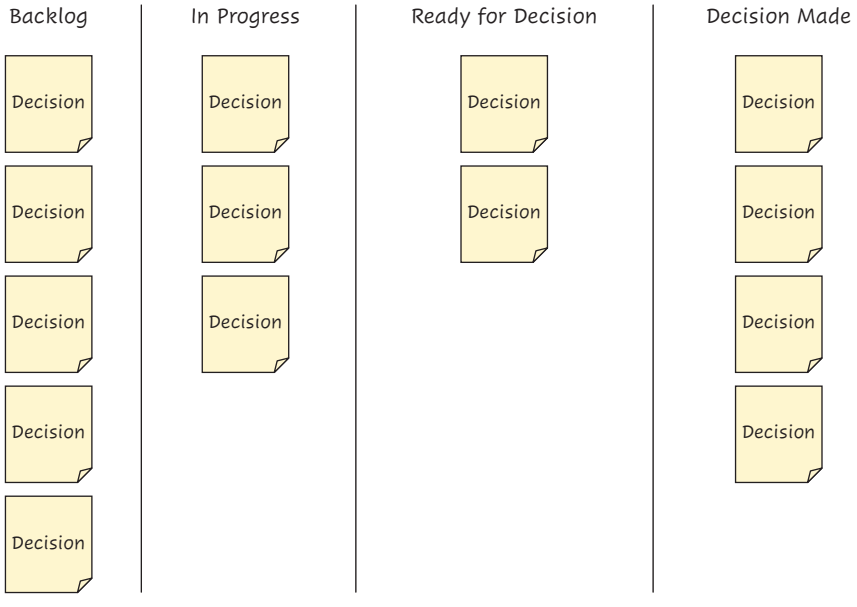


Figure 2.12 *Architectural decision Kanban board*

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

Summary

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

- Quality attributes, which represent the key cross-cutting requirements that a good architecture should address.
- Architectural decisions, which are the unit of work of architecture.
- Technical debt, the understanding and management of which is key for a sustainable architecture.
- Feedback loops, which enable us to evolve the architecture in an agile manner.

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

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

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

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

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

Index

Numerics

A

B

C

D

E

F

G

H

I

J-K

L

M

N

O

P

Q

R

S

T

U-V-W

X-Y-Z