Carnegie Mellon University
Software Engineering Institute

A Holistic View of Architecture Definition, Evolution, and Analysis

Rick Kazman
Sebastián Echeverría
James Ivers

August 2023

TECHNICAL REPORT
CMU/SEI-2023-TR-004
DOI: 10.1184/R1/20722402

Software Solutions Division

Table of Contents

List of Tables

Abstract

This report emerged from a series of technical reports, each of which analyzed a single architectural quality attribute. In this report, we take a step back from the details of any specific quality attribute and instead focus on how architectural decisions and architectural analysis spanning multiple quality attributes can be performed in a sustainable and ongoing way. This approach requires taking a holistic view of architectural activities that does not focus on a single quality attribute and that does not end when code is written or released.

It is then possible to reason about the synergies and tradeoffs among quality attributes. Synergies present an architect with unparalleled opportunities—where the choice of a single mechanism might result in benefits for multiple quality attribute concerns. In this process, architects and analysts must pay attention to the positive and negative effects of decisions since they may result in unacceptable tradeoffs that undermine system quality. This report emphasizes the importance of ongoing governance, analysis, and evaluation. To make this approach practical, it must be automated (to the extent possible); otherwise, the analysis will not be done or will not be done often enough. The report concludes with a six-step meta-playbook, a process you can use for creating your own analysis process.

1 Introduction

This is the final report in a series that represents our best understanding of how to systematically analyze an architecture in relation to a set of quality attribute requirements [Kazman 2020a, 2020b, 2022a, 2022b]. Each earlier report focuses on a single quality attribute of interest: integrability, maintainability, robustness, and extensibility.

In this report, we provide an organizational framework for reasoning about architectures, their many quality attribute requirements, and their downstream realizations in terms of more detailed design choices as well as implementation and deployment decisions. In this framework, we want to reason about the satisfaction of architectural requirements, their realizations and (potentially) their interactions—when a system is envisioned, created, grown, and maintained.

Therefore, we must consider how to do this architectural reasoning in a sustainable and ongoing way, not just at a single point in time. Since all meaningful software-intensive systems grow and evolve, we must reason in a holistic way and consider the many dimensions of concerns involved in creating, maintaining, and evolving a complex software-intensive system.

The primary goal of this report is to support reasoning around a system's quality attributes, enabling an analyst—or any motivated stakeholder—armed with the appropriate knowledge to assess the risks involved in today's architectural decisions in light of anticipated future tasks and risks, and to do so efficiently at any point in a system's lifetime. Using this lens, an architect or analyst can identify (1) opportunities for synergy among the architectural decisions being made, along with their realization in more detailed design and implementation decisions and (2) areas where tradeoffs may be necessary.

There are several commonly used and documented views of software and system architectures [Clements 2010]. The *Comprehensive Architecture Strategy*, for example, proposes four levels of architecture, each of which may be documented in terms of one or more views [Jacobs 2018]:

1. functional architecture: The Functional Architecture provides a method to document the functions or capabilities in a domain by what they do, the data they require or produce, and the behavior of the data needed to perform the function.
2. hardware architecture: A Hardware Architecture specification describes the interconnection, interaction, and relationship of computing hardware components to support specific business or technical objectives.
3. software architecture: A Software Architecture describes the relationship of software components and the way they interact to achieve specific business or technical objectives.
4. data architecture: A Data Architecture provides the language and tools necessary to create, edit, and verify Data Models. A Data Model captures the semantic content of the information exchanged.

The focus of this report—as with all of the reports in this series—is almost entirely on the *software* architecture because a software architecture is the major carrier of and enabler of a system's driving quality attributes [Bass 2021]. And since software typically changes much more quickly than hardware, it is often the primary focus of any maintenance or evolution effort. However, architectural decisions may have implications for each of the other views. In this report, we strive to embrace a holistic view of the creation and analysis of an architecture—one that spans multiple quality attributes, multiple views, multiple stakeholders (and their requirements)—and the evolution of these over time.

Even the best architecture will not ensure success if a project's governance is not well thought out and disciplined; if the developers are not properly trained; if quality assurance is not well executed; if policies, procedures, and methods are not followed; and if communication among the system's stakeholders is not fostered. Thus, we do not see architecture—even a thoughtfully designed and rigorously analyzed one—as a panacea but rather as a necessary precondition to success. This success depends on many other aspects of a project being well executed and where continuous attention is paid to the monitoring and maintenance of architectural quality.

This report describes how to combine the tools and practices we have acquired over the years to improve architecture creation, analysis, and monitoring to

1. understand the similarities and differences among quality attributes so that synergistic decisions can be made when creating the architecture
2. create a cradle-to-grave approach to determining, assessing, and maintaining architecture quality along with the quality of the realization of architectural decisions in terms of more detailed design decisions and implementations

Sidebar: Architectures Exist to Satisfy Business Goals

A system's driving quality attributes are derived from an organization's business goals for that system and architectural decisions are made to achieve those goals. But that's not the whole story. "More" of a quality attribute is not always the best way to achieve business goals, as achieving higher measures of one quality attribute may impose a high cost (e.g., additional work) or impact achievement of other quality goals (e.g., tradeoffs). Refining business goals with quality attribute scenarios to set realistic, measurable goals helps teams avoid under- or over-performing against business goals.

For example, there is a spectrum of the kinds of systems that are extensible. Where a given system lives on this spectrum is a key design decision that affects important architectural characteristics. At one extreme of this spectrum, there are "rich" systems that provide business or mission value without any extensions. Most systems created today fall into this category; they have no built-in extensibility, and all changes must be made by modifying the core.

At the other extreme, there are systems that provide little or no business or mission value without the addition of extensions (e.g., platforms). At this extreme, a system could provide just an extensible core capability, and all the functionality that users care about is achieved by extension points. Middleware platforms (e.g., node.js and the Eclipse IDE) are close to this end of the spectrum.

In between are systems that provide an important and independently useful set of functions within the core but are almost always extended in a vast number of ways. Modern browsers fall into this part of the spectrum.

Since there is no “best” point on this spectrum, a key aspect of design and analysis should be to ensure that the level of extensibility the architecture provides is properly aligned with the stakeholders’ goals for the system. These goals can be efficiently expressed with quality attribute scenarios that focus on anticipated extensions to the system and their desired characteristics.

2 Why Holism?

When creating an architecture, tradeoff decisions may be made consciously and subconsciously. Tradeoffs are inevitable; they are part of the fabric of decision making in every domain, not just software-intensive systems. But a system's tradeoffs are not always made explicit in the architecture or in later phases of more detailed design, and even when they are, the rationale for them is often not captured. As such, architectural decisions may easily be undermined as a system evolves, scales, and is maintained.

Examples of common tradeoffs include the following:

- implementation time vs. maintainability: When architecting for maintainability, it generally takes more time to create something generic and easy to modify, extend, and reuse. If short-term implementation time is more important than long-term maintenance effort, it is possible to significantly reduce this time if the system created is less flexible and more constrained. This tradeoff is sometimes termed *simplicity vs. flexibility*.
- cost and complexity vs. throughput and robustness: Architecting to allow servers to be added to a server pool can increase throughput by distributing requests across more resources and can increase availability measures (e.g., mean time to failure and mean time to repair) since failed servers can be replaced, often seamlessly, by others in the pool. However, the provision of additional servers and the infrastructure to distribute the load and check server liveness increases system cost and complexity during development and operations.
- extensibility vs. latency: Using a pattern like Observer increases latency since it adds a step of indirection between a request and a response. However, it also makes it easier to modify and, in particular, extend the system (e.g., to add new event publishers and listeners).
- maintainability vs. robustness: In general, the fewer elements and types of elements a system has, the easier it is to maintain since there is less for a newcomer to learn, fewer inter-element dependencies, and less for a maintainer to understand when making a change. However, robust systems tend to be broken down into many smaller parts (e.g., microservices), each of which is built, evolved, and deployed separately, potentially using heterogeneous technologies and environments.
- portability vs. performance: An implementation team may decide to bridge an abstraction layer (contradicting an architecture decision) to directly access the hardware to improve run-time performance; however, this decision will compromise the portability of the system, potentially inducing vendor lock-in (undermining business goals).
- implementation time vs. run-time performance: It generally takes more time to design and implement something that performs efficiently (e.g., to exhibit low latency or high throughput at scale) than to implement something that merely computes the correct response.
- latency vs. space: Response time can often be saved by using a design that pre-computes and caches some results. This is a common strategy in web browsers, databases, and online analytical processing (OLAP) cubes. The tradeoff can work in the other direction as well (e.g., using compression to save space at the cost of retrieval time).

- testability vs. latency: The more extensive the code instrumentation, the more that state and timing behavior can be analyzed and understood. However, this instrumentation increases execution time and complicates the observation of timing-related bugs.

Many of these tradeoffs are between what an implementation can support initially (e.g., meeting a specific performance requirement, such as a latency measure) and what the architecture might need to accommodate in the future or in different environments (e.g., being ported to a new platform, having its functionality extended, scaling its capabilities).

Tradeoffs such as these can and should be explicitly reasoned about when a system is being built or when it is undergoing a release, upgrade, technology refresh, refactoring, or other major event in its lifetime. At these milestones, an architectural evaluation could be undertaken, using a technique such as the Architecture Tradeoff Analysis Method (ATAM) [Clements 2001] or the Cost Benefit Analysis Method (CBAM) [Kazman 2001]. Other quality-attribute-specific analyses may also be employed to understand these tradeoffs, including tactics-based analysis, simulation, prototyping, and the construction of formal models. Several examples of these analyses are described in the Software Engineering Institute (SEI) report, *Robustness* [Kazman 2022a].

As long as a system is being maintained and evolved—and as long as its business goals are evolving—its architecture is never done. Hence, activities to update and analyze an architecture are never done, and their realization in code, frameworks, and external components is never final.

Developers always work within the context of an existing architecture, but their activities often cause that architecture to mutate into a new one that might not have been consciously envisioned, specified, or analyzed. This is how architectures erode, and it is one of the most important ways that technical debt accumulates, as Lehman first noted in his "laws" of software evolution [Lehman 1980]. As a consequence, the properties that we carefully planned and analyzed for a system may not hold as that system is maintained and evolves in response to new environments and business goals.

For example, adding filters to a processing pipeline may increase its end-to-end latency. Adding more processes to an existing processor may overload its memory, overload the CPU, or increase competition for other limited resources. Adding new modules and responsibilities to a layered architecture may compromise the cohesion of the layers.

Similarly, changes in a system's requirements, constraints, or context may change the evaluation of tradeoffs. For example, using an Observer pattern may have had a tiny effect on latency when the system was deployed on a high-end server, but it may have significant consequences if a version must be deployed on an embedded device.

To address this situation (where analysis work is never done), a holistic approach is needed to create and analyze architectures—one that does not focus on a single quality or a small number of qualities, does not simply focus on explicit architecture activities, and does not end when a release is shipped.

3 The State of the Art in Architecture Design and Analysis

When defining an architecture, an architect is tasked with making many significant, long-lived decisions. This activity cannot be entirely disconnected from its context. It can be divided into subtasks, but an architect must always maintain a vision of the whole since each part could affect the remainder in significant ways. Thus, creating an architecture is a series of *interconnected* decisions. Each decision may affect multiple quality attributes in various ways, and each decision may positively or negatively affect the expected outcomes of the other decisions. In turn, each architecture decision may require making additional subordinate architecture, design, or implementation decisions. For example, if an architecture decision is made to have the system's parts communicate in a peer-to-peer fashion, then further design or implementation decisions will be required to define the specific mechanisms that will implement that decision, the parameters that will control the communication, and the mapping onto implementation environments.

The complexities of the relationships among architecture decisions requires understanding how they actually address the needs of the system's stakeholders as well as how they may affect one another. The analysis of the architecture is then an examination of those decisions to assess the degree to which they satisfy the architectural drivers of that system. The main (but not the only) architectural drivers that will guide the decisions and their analysis are the quality attributes that the stakeholders of that system require.

3.1 The Architecture Design and Analysis Body of Knowledge

There is a rich body of knowledge that supports architecture design and analysis, some of which was referenced in the previous reports in this series. The following are several of the main concepts used:

- *Quality attribute scenarios* are quality requirements, described as a six-part scenario, that indicate how the system will react when a specific stimulus occurs and that contain a measure that allows the scenario to be evaluated. These scenarios are useful since they provide context to a quality attribute requirement and help measure whether an architecture or a system is satisfying it. These scenarios are referred to as *architectural test case*s.
- *Quality attribute characteristics* attempt to define the goals of a quality attribute more precisely and at a finer granularity. For example, extensibility is a quality attribute, and some characteristics of a system that exhibits extensibility include the loose coupling and fine-grained control of deployments. Defining quality attribute characteristics allows us to think about specific measurable aspects of the quality attribute and focus on the specific aspect of the quality attribute that is most applicable to the stakeholder's requirements.
- *Tactics* are sets of elemental choices available to an architect when making decisions to satisfy architectural drivers. They are foundational concepts that can be used as building blocks of an architecture, and hence are the raw materials from which patterns, frameworks, and styles are constructed. Tactics help when reasoning about satisfying a quality attribute or mitigating the effects of a quality attribute tradeoff to satisfy another quality attribute requirement.

- *Patterns* are conceptual solutions to recurring design problems that exist in a defined context. An architectural pattern defines a set of element types and interactions; the topological layout of the elements; and constraints on topology, element behavior, and interactions [Bass 2021]. Architectural patterns are commonly used when starting to subdivide a system; they help structure the system in a way that is appropriate to the quality attributes we want to satisfy. For example, the code modules in most non-trivial systems are structured as a set of layers, and a common way of keeping details about the user interface (UI) implementation separate from the functional core of a system is to use a pattern like model-view-controller (MVC) or model-view presenter (MVP) [Buschmann 1996].

Building to a great extent on this body of knowledge, several methods have been developed that support the creation and analysis of an architecture. These methods define a set of processes to be carried out when envisioning, creating, and evolving an architecture to help define drivers, satisfy them in a rational way, or evaluate how well they are being addressed. The following are examples of these methods:

- A *Quality Attribute Workshop (QAW)* is a method for identifying and prioritizing an organization's most critical quality attributes desired for a system by deriving them from business goals. A QAW has a defined set of steps and guidelines for who should participate and what artifacts should be provided to foster discussion. It is structured as a workshop that guides key system stakeholders to express the needs of a system in terms of its quality attributes. QAWs are useful when starting a project or when making a major change to a project to elicit the most important architectural drivers of a system.
- *Attribute-Driven Design (ADD)* is a step-by-step method for designing the architecture of a software-intensive system, basing the design process on the architecture's quality attribute requirements. It works by recursively decomposing the system and choosing tactics, patterns, and externally developed components at each iteration to satisfy the architectural drivers. ADD can be used at any point in the software development process of a system to ensure that quality attributes are being considered and addressed as needed.
- *Architecture Tradeoff Analysis Method (ATAM)* is a method for evaluating software architectures relative to their most important quality attribute goals [Clements 2001]. This method works by guiding stakeholders to express their quality attribute requirements and mapping these requirements to a representation of the architecture. The ATAM helps to surface risks that could prevent the organization from achieving its business goals. An ATAM evaluation is conducted by a trained evaluation team, along with architects and representative stakeholders. When an architecture specification or a system already exists, ATAMs are frequently used to evaluate how well it is satisfying its architectural drivers.

3.2 Supporting Design and Analysis

Other important aspects related to the creation and analysis of an architecture include tools and techniques that can assist with specific analysis tasks or measurements. The following are the most common and important tools and techniques:

- metrics: Having a quantitative measure of how well a system has satisfied a quality attribute requirement is critical to properly evaluate how a design is addressing its architecture drivers at any point in time. Metrics allow us to see how close or far from a goal a system is, and they ground architectural drivers in reality by concisely associating them to specific system requirements. Metrics can be used as leading indicators to evaluate an architecture before the system is built or as trailing indicators to collect data from the system as it runs or evolves. These metrics can also be tracked over time to understand the trajectory of the system in relation to these drivers. Both leading and trailing indicators can be useful in understanding a system's properties. Trailing indicators are usually easier to collect; leading indicators can be harder to collect, but they make it easier to understand the behavior of system early, perhaps even before implementation starts, when it is easier to change the design to address emerging problems.

- simulations: Simulations can be used to evaluate a simplified model of a system. They can be run multiple times and with different parameters, allowing the system's behavior across its parameter space to be explored without having to wait for the system to be implemented. Continuous and discrete simulations can be used to create an abstraction of a system based on its architecture and to understand the relationships between architecture parameters and the achievement of required quality attributes.
- prototyping: Creating simplified or partial implementations of a system is a relatively cheap way to evaluate how a system or one of its components will perform in relation to its quality attribute requirements. Prototyping can be critical when unknown technologies or components will be used in the design, but the exact behavior and limitations of using them are unknown. For example, if a quality attribute scenario requires a certain response time when recovering from network errors, a prototype can be created to evaluate whether those measures are achievable with the chosen technologies.
- modeling: Creating a formal model of a system can be very useful to prove whether certain requirements can be satisfied. A *formal model* is a mathematically defined abstraction of a system that can be built prior to or alongside its architecture. This model can be used to prove whether certain assertions will hold and under which conditions they will hold. Such models are often created for safety-critical systems, where the inability to ensure that an assertion is always true may lead to bad behavior, including the loss of lives. Examples of architectural models that are commonly used to analyze an architecture's robustness [Kazman 2022a] are reliability block diagrams, fault trees, Colored Petri nets, and Markov models [Boyd 1998].
- tooling: A number of tools claim to measure or assess architecture and design quality. These tools can be used to identify flaws, produce an overall health measure, and track the health of an architecture over time. Unfortunately, the empirical basis and reliability of many of these tools have been called into question, so they should be used with caution [Lefever 2021].
- quality Attribute Tactic/Concern Tables: Tables that summarize how an architecture is likely to address different quality attributes or quality attribute characteristics can provide early insight into an architect's decisions. These tables summarize how different mechanisms help or hinder quality attributes of the system. They can be a good starting point for an architect to reason about which mechanisms to select when trying to address specific quality attribute concerns. Table 1, Table 2, and Table 3 provide examples of this type of reasoning. Table 1 focuses on extensibility concerns and the tactics that address them [Kazman 2022b]. Table 2

and Table 3 focus on integrability and maintainability tactics respectively and their effects on various aspects of coupling.[1]

In the following tables, a plus sign indicates that the tactic positively addresses the characteristic and its measures, a minus sign indicates that the tactic has a negative effect, and an asterisk indicates that the tactic might positively or negatively address the measure, depending on its realization. A blank cell means that the tactic has no consistent effect on the measure.

Table 1: Extensibility Tactics and Their Relationships to Extensibility Characteristics

Tactic	Loosely Coupled	Highly Cohesive	Understand-able	State Controllable	State Observable	Granular Deployability	Controllable Deployability	Efficient Deployability
Encapsulate	+		+					
Use an intermediary	+							
Restrict communication paths	+							
Abstract common services		+	+					
Defer binding	+							
Discover service (static)	+		+					
Discover service (dynamic)	+		+					
Specialized interfaces				+	+			
Record/playback				+	+			
Localize state storage					+			
Orchestrate	+					+	+	
Manage resources	+						+	
Segment deployments							+	+

Table 2: Integrability Tactics and Their Impacts on Aspects of Coupling

Tactic	Size	Syntactic Distance	Data Semantic Distance	Behavioral Semantic Distance	Temporal Distance	Resource Distance
Encapsulate	+	+	+	+		
Use an intermediary	*	*	*	*	*	
Restrict communication paths	+					
Abstract common services	+					

[1] Since these tables provide only general guidance, other activities (e.g., strategic prototyping) are usually needed to provide more precise measures of a quality attribute, if those are measures deemed necessary.

Tactic	Size	Syntactic Distance	Data Semantic Distance	Behavioral Semantic Distance	Temporal Distance	Resource Distance
Adhere to standards	*	*	*	*	*	
Discover service (static)	+					
Discover service (dynamic)	+					
Tailor interface		+	+	*		
Configure behavior		*	*	*	*	
Orchestrate	+					
Manage resources						+

Table 3: Maintainability Tactics and Their Impacts on Aspects of Coupling

Tactic	Syntactic Distance	Data Semantic Distance	Behavioral Semantic Distance	Temporal Distance	Resource Distance
Encapsulate	+	+	+		
Use an intermediary					
Restrict dependencies					
Abstract common services	+	+	+	+	
Split module					
Increase semantic coherence					
Defer binding	+	+	+	+	+

What can we learn from these tools and techniques? In particular, how can we use them *effectively* in architecture design and analysis? This is the topic of the next section.

4 Towards a Reflective Practice of Architecture Design and Analysis

In Section 2, we discussed the ubiquity of tradeoffs in architecture design and analysis and how these tradeoffs emerge and change over time due to the normal activities of system maintenance and evolution. In Section 3, we also discussed a number of analysis methods and tools that you can use to support architectural design and analysis activities.

However, simply having methods, models, and tools is not enough for a number of reasons:

- Decision-making in a software-intensive system is continuous since development and maintenance of the system are continuous. Also, poor coding practices can undermine the best architecture; for example, code duplication (sometimes called *clone and own*) increases code size and complexity.
- Developers often make implicit implementation decisions that affect the architecture. The consequence of this behavior has been called *architecture erosion* or *architecture drift* [Whiting 2020]. Developers are often completely unaware of the consequences of their decisions. Such decisions are often the result of failing to follow the system's abstractions and conventions. For example, a 2016 case study describes how the documented module structure of Apache HDFS differed greatly from what was actually implemented [Kazman 2016]. This disparity could be the result of a failure of the architecture (e.g., perhaps it was not envisioned correctly from the start), or it could be a result of a failure of governance, where the architecture itself was fine but developers deviated from it.
- Despite their best intentions, designers and implementers are subject to cognitive biases, which can cause them to be blind to the negative consequences of their decisions and actions.
- Stakeholder communication is frequently inadequate and improperly aligned with the structure of the architecture. These communication issues can result in erosion of the conceptual integrity of the architecture [Mauerer 2022, Tamburri 2021].
- A system's context and priorities may change over time, and this may or may not be explicitly acknowledged and addressed by conscious redesign and refactoring.

Over time, every non-trivial software system evolves. Features are added, modifications are made to accommodate environmental changes, and bugs are discovered and fixed [Kazman 2020b]. In addition, some systems undergo more substantial changes to integrate with new external systems or components [Kazman 2020a]; port to a new platform; extend and scale well beyond their initial capacities [Kazman 2022b]; or improve their security, testability, or robustness [Kazman 2022a].

As these changes are made, the system's conceptual integrity (i.e., the degree to which it adheres consistently to a set of design rules [Baldwin 2000]) degrades. As a consequence, feature addition rates decrease, bug resolution time increases, and resource utilization increases. While all of this is happening, the staff turnover might increase and institutional knowledge is invariably lost. In theory, it does not have to be this way, but few systems take a different path. Jacobson calls this degradation of the architecture and its implementation *software entropy* [Jacobson 1992].

To manage and combat software entropy, we advocate a conscious and continuous practice of reflection about the state of the architecture [Razavian 2016]. For this reflection activity to be effective, it must be supported by appropriate artifacts. These artifacts could be embodied in methods or tools, but to be trusted, they must be supported by an appropriate empirical basis. This is the case in every mature engineering discipline; practicing engineers' design and analysis activities are supported by a scientific foundation on which decisions can be confidently made and tested [Shaw 2009].

In Section 5, we examine how to make and analyze decisions that span multiple quality attributes with the goal of providing synergistic benefits. In Section 6 we consider how to achieve an empirically grounded practice of continuous evaluation.

5 Architecture and Multiple Quality Attributes

When making architecture decisions, it is natural to follow a simple process:

1. Consider an architectural driver (ideally represented by a quality attribute scenario).
2. Consider candidate solutions to that driver (e.g., tactics, patterns, externally developed components).
3. Select from the candidates, and consider how to instantiate the selected mechanism.

The choice of a single mechanism, however, may affect multiple system characteristics. If the effects "point in the same direction," helping multiple quality attribute characteristics at once, the architect's choice is easy.

To illustrate this process, consider the generic notion of modular design. For example, consider the "SOLID" principles for good object-oriented design [Martin 2018]:

- single-responsibility principle: "Every class or module should have just a single responsibility, and that responsibility should be encapsulated by the class or module."
- open-closed principle: "Software entities […] should be open for extension, but closed for modification." This principle means that a class can be extended, typically through inheritance, without modifying that original class.
- Liskov substitution principle: "Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." This principle means that it should always be possible to replace a class with one of its subclasses without affecting the behavior or properties of the program.
- interface segregation principle: "Many client-specific interfaces are better than one general-purpose interface." The intent of this principle is that no client should be forced to depend on interfaces or parts of interfaces that it does not use.
- dependency inversion principle: "High-level modules should not depend on low-level modules. Both should depend on abstractions." And "[a]bstractions should not depend on details. Details should depend on abstractions."

These principles are all specific ways of achieving the more abstract characteristics of low coupling and high cohesion in a software system. Systems that are well decoupled and cohesive are, in general, easier to maintain, extend, modify, and port. Thus, creating an architecture with design rules [Baldwin 2000] that enforce SOLID principles tends to lead to those desirable outcomes. The point is to highlight that a fundamental decision in the architecture (e.g., the decision to make a class with just a single responsibility) may have benefits for multiple drivers in the system.

5.1 Synergistic Architectural Choices

Architectural decisions may be synergistic. That is, a single choice may have benefits for multiple quality attribute concerns.

Consider this example: the choice of an architectural pattern. Choosing a publish-subscribe pattern, for example, may have positive benefits for some aspects of the system's extensibility (because it is easy to add a new publisher or new subscriber for a particular topic) and for some aspects of its maintainability (because publishers and subscribers only communicate via topics that are mediated by a message broker, so they are loosely coupled, making it easier to change one without affecting the other).

This kind of reasoning can be formally traced by referring to the tactics that the publish-subscribe pattern employs, as described in Table 2 of *Extensibility* [Kazman 2022b]. Examples of such tactics are encapsulated, use an intermediary, restrict communication paths, and defer binding. In turn, these tactics support a number of desirable characteristics of the architecture, as described in Table 1 of *Extensibility*, making it more loosely coupled and understandable [Kazman 2022b]. These characteristics support aspects of other quality attributes. For example, as described in Table 2 of *Extensibility*, these same tactics support multiple characteristics that help make a system more maintainable [Kazman 2020b].

On the other hand, even when a decision has effects that go in the same direction for several quality attributes, they may be so localized that they are of little help to some of the system's quality attribute drivers. For example—continuing the example above—choosing the publish-subscribe pattern may have positive benefits for an extensibility requirement if the stakeholders need the system to easily accommodate new publishers or subscribers as it evolves. However, if this pattern was applied to, for example, the communications layer of the system, its impact will be localized in just that portion of the system.[2]

If there is a maintainability requirement on the modules used to filter data obtained by the system's sensors, the publish-subscribe pattern that was applied to the communications layer will provide no maintainability benefit to the sensor data processing modules. Similarly, if there was a requirement for portability of the layer handling the connection with specific devices, applying the publish-subscribe pattern to the communications layer would not improve the portability of that part of the system (despite the fact that, in general, publish-subscribe could be used to address that quality attribute requirement).

[2] The word *local* can have several different interpretations in the context of architecture reasoning. The first and most common interpretation is that a decision is localized to one part of the code base (e.g., a decision to use Scala to manage a data pipeline on one portion of a system or a decision to use a decorator pattern to add some functionality to a class without modifying the class itself). A second interpretation is that a decision is local to a deployment (e.g., the choice to allocate a container to a specific cloud instance). A third interpretation is that a decision has only local impact (e.g., the choice of an algorithm in one element might not appreciably affect CPU load); in that case, it is a local decision, affecting only that element. However, if the algorithm resulted in enormous CPU usage, that decision now has architectural scope.

The specific parts of the system where patterns or tactics are applied will constrain whether or not other (related) quality attributes are aided by those mechanisms. Thus, it is critical to analyze each quality attribute requirement and each mechanism choice separately, paying attention to the scope of each mechanism, to evaluate whether the architectural decisions made for one quality attribute requirement affect the others.

Sidebar: Location, Location, Location

When considering the choice of an architectural mechanism, an architect must be mindful of not just the pros, cons, and tradeoffs of that decision, but also its scope. As Eden and Kazman described in "Architecture, Design, Implementation," the Intension/Locality thesis distinguishes between architecture, design, and implementation by defining two kinds of abstraction [Eden 2003]:

- *Intensional (vs. extensional) specifications are "abstract" in the sense that they can be formally characterized by the use of logic variables that range over an unbounded domain;*
- *Non-local (vs. local) specifications are "abstract" in the sense that they pervade all parts of the system (as opposed to being limited to some part of it).*

Note that a decision might pervade all parts of the system directly (e.g., a decision to use a specific implementation platform), or it might pervade all parts of the system because of its system-wide influence, significantly affecting the use of shared resources (e.g., memory, CPU, and communication bandwidth).

Based on these definitions, Eden and Kazman stated the Intension/Locality thesis as follows [Eden 2003]:

1. *Architectural specifications are intensional and non-local;*
2. *Design specifications are intensional but local; and*
3. *Implementation specifications are both extensional and local.*

This means that the scope of a decision matters. In particular, for a decision to be "architectural," it must be non-local (i.e., its scope is across the system, where a system may, itself, contain multiple architectures).

Consider the implications of scope. Assume that an architect creating a system chose a layered pattern to organize the system's modules. Layering is a widely used pattern and promotes maintainability and extensibility in general. However, does it help with a specific maintainability scenario or a specific extensibility scenario? This is where scope comes into play. If, for example, the layering was defined to make later database replacement easier, one might expect to see a database abstraction layer, or a database abstraction module within a layer. In this case, the scope of the layering includes this anticipated evolutionary scenario and, presumably, the ripple effects of such a change would be minimized if and when the database actually needs to be replaced. In general, the layers pattern provides positive benefits for maintainability [Kazman 2020b]. If the architect did not consider this scope, then the use of a layered pattern would not provide significant maintainability benefits for this change.

Consider another example. Perhaps the layering pattern was chosen for extensibility purposes. As stated in *Extensibility* [Kazman 2022b], "Layers allow a clear separation of the core system and extension points where new planned functionalities can be added later." However, if the system needed to be extended by adding a new sensor and this kind of addition was not anticipated or provided for in the layering (i.e., that is, if the scope of the layering did not include this extension point), the putative benefits of layering would be lost.

An architecture must, therefore, be analyzed not only in terms of the mechanisms chosen or not chosen, but also in terms of the appropriateness of the scope of those mechanisms. Scenario-based analysis can be helpful in this exploration of appropriateness.

5.2 Tradeoffs Among Decisions

Sometimes an architectural decision and subsequent related detailed design and implementation decisions may be even more of a challenge for the architect if the effects on key characteristics go in different directions; this is a tradeoff, as discussed in Section 2. Continuing the example from Section 5.1, if a publish-subscribe pattern was chosen and its implementation employs a message broker, this implementation may be a point of resource contention in the executing system—a common concern when employing this pattern [Sena 2018]. Therefore, as large numbers of publishers, subscribers, and topics are added to the system over its lifecycle, worst-case latency may suffer [Bass 2021].

This risk is not inherent in the publish-subscribe pattern, but it may occur when the pattern is realized with specific technologies. Of course, other tactics may be employed to address this tradeoff. In practice, many highly scalable brokers have been created[3] through the use of performance tactics [Bass 2021]; examples include increase resources, introduce concurrency, maintain multiple copies of computations, and reduce computational overhead.

Furthermore, the most pernicious tradeoffs are frequently between cost, benefit, and schedule, rather than just "technical" quality attributes. As a result, these aspects must also be considered in the design decision, otherwise it is not holistic [Carriere 2010].

For these reasons, we advise architects to think across quality attribute requirements rather than focus on a single requirement at a time. Every non-trivial system has multiple quality attribute requirements across different qualities, and we want to take advantage of opportunities for synergy whenever and wherever possible.

When refined as quality attribute scenarios, which are good ways to make the quality attributes measurable, each scenario provides an architect with indications of which characteristics of the relevant quality attributes (e.g., semantic distance) are important and where in the architecture they are relevant (e.g., elements involved in the scenario). Not every quality attribute scenario will require a single (new) mechanism. The tables in the other reports in this series can provide insight into how to make architectural decisions that affect multiple quality attributes at once, whether they help you achieve multiple drivers or whether they are design tradeoffs [Kazman 2020a, 2020b, 2022a, 2022b].

[3] See "A highly resilient and scalable broker architecture for IoT applications" [Sen 2018].

Looking across quality attribute requirements and how they are satisfied by chosen mechanisms in the architecture can provide architects with a valuable gestalt view. The following are a few simple examples of how such reasoning might work:

- These 10 quality attribute requirements all involve the same sections of the architecture. This suggests that there are some common decisions that are worth considering, but tradeoffs at different levels may also be likely.
- These 5 quality attribute requirements all involve the same characteristics with respect to the same elements; hence, common decisions are highly desirable.
- These 8 quality attribute requirements involve the same characteristics, but they affect different sections of the architecture. In this case, common decisions are not necessary since decisions made for one element will not help or hurt decisions made for others.

These decisions consider location and are always expressed in terms of whether or not the elements affected by architectural decisions are common.

6 Towards Continuous Evaluation

Throughout the history of software development, there has been a clear trend towards an increased velocity of system changes and system deployments. The Spiral Model [Boehm 1988], Rapid Application Development [Martin 1991], Lean and Agile practices, and the DevOps movement [Bass 2015] are all instances of this trend over the decades. Today, this trend continues with an increasing emphasis on dynamic and adaptive architectures [Salehie 2009, Sobhy 2022] that dynamically adapt to changes in demand, resources, or their environment.

Given this context, it is obvious that analysis, for the vast majority of modern systems, is not "one and done," just as (and, in fact, *because*) design, development, and deployment are continuous. However, traditionally, architectural analysis has been human centric, requiring skilled trained analysts, requiring the provision of documentation, and typically involving multiple meetings with the architect and other important stakeholders [Clements 2001, Kazman 2002].

This continuous process, while potentially valuable for many reasons (e.g., improved stakeholder communications, clarified requirements, improved architectures, identification of risks), is time consuming, and the inherent cost and scheduling complexities of such meetings make them difficult to conduct in practice. Because these meetings involve many stakeholders, if and when they are scheduled, they will inevitably happen infrequently [Erder 2015]. In most cases, this approach clearly does not match the cadence of the projects.

How do we resolve this apparent paradox? There is a tension between the desire for engineering control over the qualities of our systems and our desire for rapid implementation and release of new functionality. In our view, the only practical alternative to frequent human-centric reviews is *in*frequent human-centric reviews enhanced by continuous automated evaluation, which should ideally feed regular reports or dashboards.

To achieve this vision of continuous evaluation, we need to change our software development practices. In particular, we must keep appropriate measures in mind when we plan and create architectures, develop them, and maintain them. This approach requires thinking differently about several aspects of how to acquire, design, and create systems, including

- planning for data collection
- tracking data at run-time
- tracking data at development time
- automating architecture analysis activities

We discuss these topics in Sections 6.1–6.4.

6.1 Planning for Data Collection

Data on both run-time and development-time characteristics must be tracked, and the architecture, system's run-time infrastructure, and development-time infrastructure should be designed to track both sets of characteristics. This kind of data collection and tracking requires planning in terms of how systems are built, how to choose what data to collect, and how to analyze it. It also requires planning for training teams to collect, analyze, and respond to the results of the analysis. While some of these practices are used in projects today (e.g., systems designed with run-time observability in mind), they are the exception and not the rule.

Thus, planning for data collection must be a conscious choice early in a system's lifetime. Since it represents a change in how systems are envisioned, built, and managed, this planning must be supported at all levels of a project, from top management to architects, designers, developers, maintainers, and quality assurance personnel.

6.2 Tracking Data at Run-Time

Observability and monitoring of run-time characteristics (e.g., latency, mean-time-to-failure, message loss, throughput) should be designed in from the start. Tools, frameworks, and a run-time infrastructure (e.g., cloud infrastructure) can enhance observability and monitoring, particularly if the characteristics being observed are common (e.g., messages, transactions, memory usage, or CPU usage). If more idiosyncratic, system-specific characteristics need to be monitored and observed, data collection and tracking must be planned and implemented, developers must be trained in their use, and code reviews should confirm their use on a regular basis.

6.3 Tracking Data at Development Time

Orthogonal to the concerns of tracking data at run-time, it is also necessary to track development-time data. To achieve agility, systems should be maintainable, integrable, robust, extensible, etc. Achieving these quality attributes implies that new features can be added to systems and bugs can be fixed quickly, efficiently, and predictably. Gaining insight into these qualities means extracting data from revision control systems, issue-trackers, and other project data sources (e.g., discussion forums, email lists), and data must be tracked over time.

Again, tracking may require process changes among the development staff. For example, teams should standardize how they use issue trackers and revision control systems. This standardization must include processes for how issues are entered, labelled, assigned, and resolved. It should also include processes for how code is checked in and committed (including ensuring that every commit is associated with an issue that the commit resolves). Without this process, it is impossible to know with certainty why commits were made, making it challenging to analyze the root causes of bugs or changes in velocity.

6.4 Automating Architecture Analysis Activities

Seemingly innocent code modifications and bug fixes can, cumulatively and over time, undermine architectural integrity [de Silva 2012]. This phenomenon has been termed *architecture erosion* or *architecture drift*. Given that this erosion contributes to increasing technical debt, it should be avoided or mitigated to the greatest extent possible.

Therefore, it makes sense to conduct analysis continuously, ideally after each change to a system (e.g., after each check-in or commit to a code repository). However, as stated above, this is infeasible if the analysis is human-intensive; no project manager would agree to do an architecture review at such a cadence. Hence, we must consider automatic or semi-automatic ways to determine if, when, and where an architecture's integrity is being undermined. This approach requires computation, such as that provided by analysis tools.[4]

There are obvious benefits of automation: reduced human effort and a reduced need for continuous strict attention to enormous numbers of details. Other advantages of automation include traceability, the ability to create continuously updated management dashboards, the ability to mine data to drive organizational learning, and the ability to flag potential problems before they become full-fledged technical debt.

[4] For examples of analysis tools, see "Detecting the Locations and Predicting the Costs of Compound Architectural Debts" [Xiao 2022], "Architecture Anti-Patterns: Automatically Detectable Violations of Design Principles" [Mo 2021], and "Software Archinaut: A Tool to Understand Architecture, Identify Technical Debt Hotspots and Manage Evolution" [Cervantes 2020].

7 Towards a Meta Playbook

To continuously evaluate the state of an architecture in a system in relation to its business goals and expected behaviors, a systematic approach is required. This implies that the process itself needs to be monitored, both while defining an architecture and later, while the system is being implemented and maintained. However, if the process that an architect and developers follow is not systematic and does not provide measurable information, it will be difficult to ensure that the monitoring process itself will generate any meaningful output.

Thus, when defining how to monitor the evolution of an architecture, one of the first things to consider is the human aspect and how different human factors affect the process itself [Tang 2017, Kazman 2002]. For example, software architects may make decisions without having elucidated an explicit rationale for them; in fact, sometimes they are not even aware they are making a major decision. Biases can also affect the reasoning behind different decisions.

Therefore, the first step is to ensure that the development process anticipates and compensates for these problems by following a step-by-step guide that

- helps make architecturally relevant decisions explicit
- guides architects to describe rationale explicitly, using a consistent structure

One way of accomplishing this is to use the playbooks we defined in previous reports [Kazman 2020a, 2020b, 2022a, 2022b].

Human factors also influence the monitoring and analysis process. Techniques similar to the ones described in our previous reports can be used to monitor the development process. In this way, a "meta-playbook" that guides the steps for monitoring the development process would be a useful starting point to ensure that the monitoring process

- is executed in a repeatable way
- follows steps that allow those monitoring its evolution to explicitly indicate the rationale for the different evaluations and analyses that are being performed

It is then important to consider the business case for continuous monitoring and analysis. The effort and effects of implementing this continuous process are not trivial. Even if the goal makes it worthwhile, it is critical to look ahead and consider the costs associated with developing and implementing such a process. Some of the costs include the following:

- up-front costs: These include all costs needed to initiate the continuous monitoring and analysis process. Activities needed to make this process work include training the people who will be involved, including architects and developers; preparing all tools, artifacts, and additional materials (e.g., data being tracked, as described in Section 5); and the additional infrastructure needed (e.g., additional servers to store collected data).
- ongoing costs: These include ongoing costs for maintaining the monitoring process and responding to any disruptions to normal development activities.

- effects on velocity: Executing a parallel process requires time from people involved in the development process. This will affect the team's velocity, both positively and negatively. Team members will better understand the status of the architecture, but they must devote time to gather metrics and analyze the data. If this evaluation is performed correctly and supported by appropriate data and tools, the net benefit to team velocity will be strongly positive.
- effects on risk exposure: While this approach to continuous evaluation is expected to result in a net benefit, following the continuous monitoring process may cause problems, particularly at the start, when the new processes and infrastructure are being implemented. For example, development iterations might take longer, possibly resulting in missed deadlines. The effects of altering a development process to include continuous monitoring must be carefully weighed to mitigate secondary issues.
- effects on project morale: Team members may perceive the addition of a monitoring process positively or negatively, affecting morale. Depending on the team, organization, and how the process is implemented, a new monitoring process could be perceived as overhead that prevents people from focusing on what they think is most important, or it could be seen as a sign of commitment and proper coordination between different roles involved in system development. Regardless, to ensure broad buy-in, the effects on the team's morale must be considered, and the process must be clearly explained and motivated.

As discussed in Section 5, one critical activity required to follow a continuous monitoring and analysis process for an architecture is to infuse observability into the software development process and the software product, covering both run-time and development-time characteristics. This includes several different types of related activities:

- Tasks and tools will be required to easily gather metrics about the following:
  - effort spent in different steps of the development
  - effort required to ensure the architecture is still consistent
  - the status of the architectural goals in relation to the current implementation
- The system itself may need specific changes to enable it to gather run-time metrics that allow the monitors to evaluate whether run-time quality attributes are behaving as expected as new versions of the system are produced.
- Reports or dashboards must be designed and generated on a regular basis, which may include automated alerts.

This setup for metric gathering and analysis must be defined early in the development process so that it can be properly integrated, and all of the effort required for it can be properly planned along with the traditional development tasks.

To create a more detailed meta-playbook for continuous architectural evaluation, all the factors described above should be considered. This means that, while a generic meta-playbook can be developed in detail, it must be customized for the specific drivers, norms, and context for each organization that wants to use it. Regardless, it should be possible to define a meta-playbook that is less detailed than those described in previous reports to act as an overall framework of main steps and activities to consider before defining a more detailed one. The following is a summary of the main steps that such a meta-playbook could include:

- Step 1: Collect information about the current state of the architecture, design, and the process. This step is necessary to understand the environment where the process will run. This understanding could be achieved by mining documentation if it is complete and up-to-date, or through developer interviews or reverse engineering if the documentation is inadequate.
- Step 2: Define the scope of the continuous monitoring that will be used for the project. Understanding the types of drivers that must be monitored is critical in defining how much of the process to actually attempt to regularize and, ideally, automate. This will define how complex the process can be, how much tool support it will require, and the technical aspects it will focus on.
- Step 3: Evaluate the costs for the defined scope. Once an initial scope is specified, the different costs described above can be estimated to determine how much effort and money will be needed to set up the process. If the evaluation of cost is not satisfactory, it may be necessary to go back to Step 2 and reassess the scope. Some experimentation and prototyping may be required to better understand the scope.
- Step 4: Define the specific changes to the current development process and the system itself that are necessary to implement the monitoring process. As described above, tools, reports, and changes to the system may be needed to implement the required process. Also, project stakeholders may need documentation, training, or other forms of coaching and guidance. Define these specific changes in advance so that they can be incorporated into the development process and the specification of the system itself.
- Step 5: Implement the monitoring process, including setting up all tools and infrastructure, following any new processes or tasks needed for the monitoring process, and ensuring all required metrics are being captured.
- Step 6: Use the process and the gathered metrics to evaluate the status of the architecture and take corrective measures as appropriate. The nature of these corrective measures should ideally be defined as part of the process itself. This last step is a recurring one that can be expanded into more sub-steps as desired and as defined in Step 2.

8 Conclusions

This report emerged from a series of four technical reports that each focused on the analysis of a single architectural quality attribute. In this report, we step back from the details of these quality attributes and instead focus on the overall picture of how architectural decisions and analysis spanning multiple quality attributes can be performed in a sustainable and ongoing way. This holistic view of architectural activities does not focus on any single quality attribute and does not end when a release is shipped.

An advantage of this holistic view is that we can begin to reason about synergies and tradeoffs between tactics and patterns across quality attributes. This view informs our architectural choices and how those choices are realized by downstream activities. Synergies represent unparalleled opportunities for an architect—where the choice of a single mechanism might result in benefits for multiple quality attribute concerns. Therefore, when making architectural choices, architects and analysts should pay attention to these opportunities. As part of their decision-making process, they should take note of both the positive and negative effects of architectural decisions and their realizations since they can result in unacceptable tradeoffs. This means paying attention to both architectural and governance practices.

Furthermore, we stress that architecture analysis and evaluation should ideally be continuous. To make continuous analysis and evaluation practical, it must be as automated as possible; otherwise, it will either not occur or will not occur often enough.

Finally, we provided a six-step meta-playbook, which provides you with a process for creating your own analysis process. The steps are generic and must be tailored to your organization's risk, needs, and anticipated benefits.

9 Further Reading

This report was inspired by the four reports about the analysis of architectural quality attributes: *Integrability* [Kazman 2020a], *Maintainability* [Kazman 2020b], *Robustness* [Kazman 2022a]. and *Extensibility* [Kazman 2022b]. The ideas developed in this report are also rooted in the broader practice of architecture analysis and design, which has been maturing in industry, government, and academia for the past three decades.

This report was also inspired by work in architecture knowledge management [Capilla 2016], reflective thinking [Razavian 2016], and prior research into the automation of architectural analysis. For deeper insights into this automation approach, consult the work of Cervantes, Xiao, and Paradis and their colleagues [Cervantes 2020, Xiao 2022, Paradis 2021].

References

[Baldwin 2000]
Baldwin, C. & Clark, K. *Design Rules, Volume 1: The Power of Modularity*. MIT Press. 2000. ISBN 9780262538206. https://mitpress.mit.edu/9780262538206/design-rules/

[Bass 2015]
Bass, L.; Weber, I.; & Zhu, L. *DevOps: A Software Architect's Perspective.* Addison-Wesley. 2015. ISBN 978-0134049847. https://www.pearson.com/store/p/devops-a-software-architect-s-perspective/P200000009378/9780134049847

[Bass 2021]
Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice, 4th Edition*. Addison-Wesley. 2021. ISBN 978-0136886099. https://www.pearson.com/en-us/subject-catalog/p/Bass-Software-Architecture-in-Practice-4th-Edition/P200000000111/9780137468218

[Boehm 1988]
Boehm, B. A Spiral Model of Software Development. *IEEE Computer.* Volume 21. Issue 5. May 1988. Page 61. https://dl.acm.org/doi/10.1109/2.59

[Boyd 1998]
Boyd, M. & Lau, S. *Introduction to Markov Modeling: Concepts and Uses.* NASA. 1998. https://ntrs.nasa.gov/search.jsp?R=20020050518

[Buschmann 1996]
Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; & Stal, M. *Pattern-Oriented Software Architecture.* Wiley. 1996. ISBN 978-0-471-95869-7. https://www.wiley.com/en-us/Pattern+Oriented+Software+Architecture,+Volume+1,+A+System+of+Patterns-p-9780471958697

[Capilla 2016]
Capilla, R.; Jansen, A.; Tang, A.; Avgeriou, P.; & Ali Babar, M. 10 Years of Software Architecture Knowledge Management. *Journal of Systems and Software.* Volume 116. Issue C. 2016. Page 191. https://dl.acm.org/doi/10.1016/j.jss.2015.08.054

[Carriere 2010]
Carriere, J.; Kazman, R.; & Ozkaya, I. A Cost-Benefit Framework for Making Architectural Decisions in a Business Context. *Proceedings of the 32nd International Conference on Software Engineering.* Volume 2. May 2010. Page 149. https://dl.acm.org/doi/10.1145/1810295.1810317

[Cervantes 2020]
Cervantes, H. & Kazman, R. Software Archinaut: A Tool to Understand Architecture, Identify Technical Debt Hotspots, and Manage Evolution. *Proceedings of the International Conference on Technical Debt, 2020.* June 2020. Page 115. https://dl.acm.org/doi/10.1145/3387906.3388633

[Clements 2001]
Clements, P.; Kazman, R.; & Klein, M. *Evaluating Software Architectures: Methods and Case Studies.* Addison-Wesley. 2001. ISBN 978-0201704822. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=30698

[Clements 2010]
Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Merson, P.; Nord, R.; & Stafford, J. *Documenting Software Architectures: Views and Beyond*, *2nd Edition*. Addison-Wesley. 2010. ISBN 978-0321552686. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=30386

[de Silva 2012]
de Silva L. & Balasubramaniam D. Controlling Software Architecture Erosion: a Survey. *Journal of Systems and Software*. Volume 85. Issue 1. January 2012. Page 132. https://doi.org/10.1016/j.jss.2011.07.036

[Eden 2003]
Eden, A. & Kazman, R. Architecture, Design, and Implementation. *Proceedings of the 25th International Conference on Software Engineering.* May 2003. Page 149. https://dl.acm.org/doi/10.5555/776816.776835

[Erder 2015]
Erder, M. & Pureur, P. *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World*. Morgan Kaufmann. 2015. ISBN 978-0128032848. https://doi.org/10.1016/C2014-0-02435-5

[Jacobs 2018]
Jacobs, W.; Wigginton, S.; & Padilla, M. *Comprehensive Architecture Strategy (CAS)*. Public release #4557. Defense Technical Information Center. 2018. https://apps.dtic.mil/sti/citations/AD1103295

[Jacobson 1992]
Jacobson, I. *Object-Oriented Software Engineering*. Addison-Wesley. 1992. ISBN 978-0201544350. https://www.ivarjacobson.com/publications/books/object-oriented-software-engineering-book

[Kazman 2001]
Kazman, R.; Asundi, J.; & Klein, M. Quantifying the Costs and Benefits of Architectural Decisions. *Proceedings of the 23rd International Conference on Software Engineering (ICSE 23).* July 2001. Page 297. https://dl.acm.org/doi/10.5555/381473.381504

[Kazman 2002]
Kazman, R. & Bass, L. Making Architecture Reviews Work in the Real World. *IEEE Software.* Volume 19. Issue 1. January/February 2002. Page 67. https://doi.org/10.1109/52.976943

[Kazman 2016]
Kazman, R.; Goldenson, D.; Monarch, I.; Nichols, W.; & Valetto, G. Evaluating the Effects of Architectural Documentation: A Case Study of a Large Scale Open Source Project. *IEEE Transactions on Software Engineering.* Volume 42. Issue 3. March 1, 2016. Page 220. https://doi.org/10.1109/TSE.2015.2465387

[Kazman 2020a]
Kazman, R.; Bianco, P.; Ivers, J.; & Klein, J. *Integrability.* CMU/SEI-2020-TR-001. Software Engineering Institute, Carnegie Mellon University. 2020. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=637375

[Kazman 2020b]
Kazman, R.; Bianco, P.; Ivers, J.; & Klein, J. *Maintainability*. CMU/SEI-2020-TR-006. Software Engineering Institute, Carnegie Mellon University. 2020. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=650480

[Kazman 2022a]
Kazman, R.; Bianco, P.; Echeverria, S.; & Ivers, J. *Robustness*. CMU/SEI-2022-TR-004. Software Engineering Institute, Carnegie Mellon University. 2022. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=875736

[Kazman 2022b]
Kazman, R.; Echeverria, S.; & Ivers, J. *Extensibility*. CMU/SEI-2022-TR-002. Software Engineering Institute, Carnegie Mellon University. 2022. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=876971

[Lefever 2021]
Lefever, J.; Cai, Y.; Cervantes, H.; Kazman, R.; & Fang, H. On the Lack of Consensus Among Technical Debt Detection Tools. *Proceedings of the 43rd International Conference on Software Engineering (ICSE) 2021.* May 2021. Page 121. https://doi.org/10.1109/ICSE-SEIP52600.2021.00021

[Lehman 1980]
Lehman, M. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*. Volume 68. Issue 9. September 1980. Page 1060. https://doi.org/10.1109/PROC.1980.11805

[Martin 1991]
Martin, J. *Rapid Application Development.* Macmillan. 1991. https://dl.acm.org/doi/10.5555/103275

[Martin 2018]
Martin, R. *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Pearson. 2018. ISBN 978-0134494166. https://www.pearson.com/en-us/subject-catalog/p/clean-architecture-a-craftsmans-guide-to-software-structure-and-design/P200000009528/9780134494166

[Mauerer 2022]
Mauerer, W.; Joblin, M.; Tamburri, D.; Paradis, C.; Kazman, R.; & Apel, S. In Search of Socio-Technical Congruence: A Large-Scale Longitudinal Study. *IEEE Transactions on Software Engineering*. Volume 48. August 2022. Page 3159. https://doi.ieeecomputersociety.org/10.1109/TSE.2021.3082074

[Mo 2021]
Mo, R.; Cai, Y.; Kazman, R.; Xiao, L.; & Feng, Q. Architecture Anti-Patterns: Automatically Detectable Violations of Design Principles. *IEEE Transactions on Software Engineering.* Volume 47. Issue 5. May 2021. Page 1008. https://doi.org/10.1109/TSE.2019.2910856

[Padilla 2019]
Padilla, M.; Davis, J.; & Jacobs, W. Comprehensive Architecture Strategy (CAS). *The Open Group*. September 2019. https://www.opengroup.us/face/documents.php?action=show&dcat=87&gdid=21082

[Paradis 2021]
Paradis, C. & Kazman, R. Design Choices in Building an MSR Tool: The Case of Kaiaulu. *1st International Workshop on Mining Software Repositories for Software Architecture at the 15th European Conference on Software Architecture.* September 13-17, 2021. http://ceur-ws.org/Vol-2978/msr4sa-paper1.pdf

[Razavian 2016]
Razavian, M.; Tang, A.; Capilla, R.; & Lago, P. In Two Minds: How Reflections Influence Software Design Thinking. *Journal of Software Evolution and Process.* Volume 28. Issue 6. June 2016. Page 394. http://dx.doi.org/10.1002/smr.1776

[Salehie 2009]
Salehie, M. & Tahvildari, L. Self-Adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems.* Volume 4. Issue 2. May 2009. Page 1. https://doi.org/10.1145/1516533.1516538

[Sen 2018]
Sen S. & Balasubramanian, A. A Highly Resilient and Scalable Broker Architecture for IoT Applications. *10th International Conference on Communication Systems & Networks.* April 2018. Page 336. https://doi.org/10.1109/COMSNETS.2018.8328216

[Sena 2018]
Sena, B.; Garces, L.; Allian, A.; & Nakagawa, E. Investigating the Applicability of Architectural Patterns in Big Data Systems. *Proceedings of the 25th Conference on Pattern Languages of Programs.* October 2018. Page 1. https://dl.acm.org/doi/abs/10.5555/3373669.3373677

[Shaw 2009]
Shaw, M. Continuing Prospects for an Engineering Discipline of Software. *IEEE Software.* Volume 26. Issue 6. November/December 2009. Page 64. https://doi.org/10.1109/MS.2009.172

[Sobhy 2022]
Sobhy, D.; Minku, L.; Bahsoon, R.; & Kazman, R. Continuous and Proactive Software Architecture Evaluation: An IoT Case. *ACM Transactions on Software Engineering and Methodology.* Volume 31. Issue 3. July 2022. Page 1. https://doi.org/10.1145/3492762

[Tamburri 2021]
Tamburri, D.; Palomba, F.; & Kazman, R. Exploring Community Smells in Open-Source: An Automated Approach. *IEEE Transactions on Software Engineering*. Volume 47. Issue 3. March 2021. Page 630. https://doi.org/10.1109/TSE.2019.2901490

[Tang 2017]
Tang, A.; Razavian, M.; Paech, B.; & Hesse, T. Human Aspects in Software Architecture Decision Making. *Proceedings of the IEEE International Conference on Software Architecture (ICSA 2017).* May 2017. Page 107. https://doi.org/10.1109/ICSA.2017.15

[Whiting 2020]
Whiting, E. & Andrews, S. Drift and Erosion in Software Architecture: Summary and Prevention Strategies. *Proceedings of the 2020 the 4th International Conference on Information System and Data Mining* (*ICISDM 2020*). May 2020. Page 132. https://doi.org/10.1145/3404663.3404665

[Xiao 2022]
Xiao, L.; Kazman, R.; Cai, Y.; Mo, R.; & Feng, Q. Detecting the Locations and Predicting the Costs of Compound Architectural Debts. *IEEE Transactions on Software Engineering.* August 2022. https://doi.org/10.1109/TSE.2021.3102221

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. **AGENCY USE ONLY** (Leave Blank)	2. **REPORT DATE** August 2023	3. **REPORT TYPE AND DATES COVERED** Final
4. **TITLE AND SUBTITLE** A Holistic View of Architecture Definition, Evolution, and Analysis		5. **FUNDING NUMBERS** FA8702-15-D-0002
6. **AUTHOR(S)** Rick Kazman, Sebastian Echeverria, James Ivers		
7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213		8. **PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI-2023-TR-004
9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** SEI Administrative Agent AFLCMC/AZS 5 Eglin Street Hanscom AFB, MA 01731-2100		10. **SPONSORING/MONITORING AGENCY REPORT NUMBER** n/a
11. **SUPPLEMENTARY NOTES**		
12A **DISTRIBUTION/AVAILABILITY STATEMENT** Unclassified/Unlimited, DTIC, NTIS		12B **DISTRIBUTION CODE**

13. **ABSTRACT (MAXIMUM 200 WORDS)**

This report emerged from a series of technical reports, each of which analyzed a single architectural quality attribute. In this report, we take a step back from the details of any specific quality attribute and instead focus on how architectural decisions and architectural analysis spanning multiple quality attributes can be performed in a sustainable and ongoing way. This approach requires taking a holistic view of architectural activities that does not focus on a single quality attribute and that does not end when code is written or released.

It is then possible to reason about the synergies and tradeoffs among quality attributes. Synergies present an architect with unparalleled opportunities—where the choice of a single mechanism might result in benefits for multiple quality attribute concerns. In this process, architects and analysts must pay attention to the positive and negative effects of decisions since they may result in unacceptable tradeoffs compromise system quality. This report emphasizes the importance of ongoing governance, analysis, and evaluation. To make this approach practical, it must be automated (to the extent possible), otherwise the analysis will not be done or will not be done often enough. The report concludes with a six-step meta-playbook, a process you can use for creating your own analysis process.

14. **SUBJECT TERMS** architecture analysis, extensibility, quality attributes, quality attribute requirements, software architecture			15. **NUMBER OF PAGES** 37
16. **PRICE CODE**			
17. **SECURITY CLASSIFICATION OF REPORT** Unclassified	18. **SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	19. **SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	20. **LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102