1.1 Introduction

Architects regularly make architectural design decisions but are usually unable to evaluate the economic impact of those decisions. Management, in contrast, is often interested in product-level decisions (such as features and quality) but not in the technical details of how those decisions are made. These differing interests lead to inconsistencies between how managers view value and how architects can enable or disable those value propositions through their design decisions. This lack of communication can result in poor decisions.

Clearly, it is in the best interests of all project stakeholders to make informed and technically feasible value-driven design decisions. Thus architects need practical, validated tools and techniques for applying economics-driven principles to software architecture. They need these tools and techniques to make better decisions and to better justify those decisions to their stakeholders.

Current practices in architecting software systems do not often make economic and strategic considerations explicit. Architectural design decisions tend to be driven by ways that are not connected to, and usually not optimal for, value creation. Factors such as flexibility, time to market, cost, and risk reduction often have high impact on value creation (Sullivan et al., 2001). But the sad reality is that such considerations are seldom included in architectural planning. The state of the practice is that systems are typically designed by thinking about functionality first and considering architecture, if at all, only when problems arise.

Such ignorance is in stark contrast to the motivation and objectives of architecture-centric approaches to development and systematic evolution (e.g., model-driven architectures, domain-specific software architectures and product lines, component-based and middleware-induced architectures, etc.), where managing complexity, cost reduction, risk mitigation, evolvability, strategic planning, and long-term value creation are among the major drivers for adopting such approaches. This suggests an urgent need for economics-driven and value-based architectural models and metrics, which can provide the architect with insights into the long-term and strategic viability, cost-effectiveness, and sustainability of the architecture design decisions made (or not made).

We believe that these metrics and models should be at the heart of the “architecting” process: inception, elaboration, composition, evaluation and analysis, implementation and deployment,
maintenance, and evolution of architectures. Such a perspective is important; it assists the objective
assessment of the lifetime costs and benefits of architectural decisions made for evolving systems.
It also helps to identify legacy situations, where an architecture or a component is indispensable
but can no longer be evolved to meet changing needs at a reasonable, sustainable cost. Such consider-
ations will form the scientific foundation for reasoning about the economics of quality attribute
requirements and project requirements in the context of architectures and architecting.

This book seeks to gather together a coherent set of research and experience on the following
topics.

- Architecture-based economic models, particularly cost modeling and benefit modeling
- Architecture and its relationship to project inception and evolution
- Economic aspects of architecture-based project management
- Architecture and its effects on risk management, particularly as it relates to technical debt
- Architecture and agility
- Tools and techniques for analyzing the economic implications of architectural decisions

1.2 Architecture and project management

Architecture is most commonly used in medium to large-scale projects—projects that have multiple
teams and too much complexity for any individual to fully comprehend. Such projects typically
involve substantial investments and have multiyear durations. For such projects, the various teams
(which may be distributed) need to coordinate their efforts. In these projects, upper management
often demands both short time to market and adequate oversight. Thus the use of architecture as a
tool for project management is becoming increasingly necessary and commonplace.

The project manager is responsible for the business side of the project. This responsibility
involves providing appropriate and sufficient resources to the project; negotiating, creating, and
overseeing the budget and schedule; negotiating requirements and constraints with stakeholders;
and ensuring overall project quality. The software architect, working closely with the project man-
ger, is responsible for the technical side of the project. This involves: designing to achieve specific
quality attributes, determining and enacting appropriate technical oversight, reviewing requirements
for feasibility, and day-to-day leadership of the development team.

Once an architecture has been designed, it can be used to define the project structure and orga-
nization. The structure of a project and the structure of the team that realizes the project need to be
reasonably congruent. This is known as Conways Law (Conway, 1968) or the Mirroring
Hypothesis (Colfer and Baldwin, 2010). Once implementation of a project commences, the project
manager and architect have a series of decisions to make involving trade-offs, incremental develop-
ment, and risk management. The project manager and architect need to negotiate any trade-offs,
particularly those arising from new requirements, as these will affect one or more of cost, schedule,
and quality. The project plan also defines the schedule, which has enormous implications for the
architecture since it has to accommodate internal “releases” of subsets of the project’s functionality,
leading up to a (hopefully) successful complete implementation. And through it all the project man-
ger must track progress. This tends to occur through some combination of personal contact with
developers (which does not scale up well), through formal status meetings, through the collection
of metrics, and through risk management.
1.3 Architecture-based economic modeling

An economic model in any domain is a simplification—an abstraction. For economic modeling to be tractable, the modeler must ignore many details of the domain being modeled and choose the most salient details to model, all the while maintaining as high a degree of fidelity as possible. Every model is an approximation, and the better the approximation the more useful the model is, all other things being equal. But all other things are never equal. They must be based on readily available, traceable, and verifiable data; they must be computationally tractable; and they must make sense to the model’s stakeholders.

An architecture is an ideal place to bring cost modeling (as well as benefit modeling, risk management, and project management, as we will see). The architecture forms a large part of a software-intensive project’s work breakdown structure and so it guides the architect and developers in doing both top-down and bottom-up cost estimates.

For example, Dan Paulish and colleagues at Siemens Corporation have developed a number of rules of thumb for doing top-down estimation of project costs (Paulish, 2002). Some rules of thumb that they have used in medium-sized (~150 K SLOC) projects are:

- Number of components to be estimated ~150
- Paper design time per component ~4 hours
- Times between engineering releases ~8 weeks
- Overall project development allocation
  - 40% design—5% architectural, 35% detailed
  - 20% coding
  - 40% testing

Once an initial architecture design has been produced, then leads for the pieces of the architecture are assigned and the teams can be built. At this point, bottom-up estimates from the team leads can be produced. This bottom-up estimate is usually much more accurate than the top-down estimate, but there may also be significant differences due to differing assumptions. These differences between the top-down and bottom-up estimates need to be discussed and negotiated. An experienced project manager knows to avoid committing costs to upper management until the bottom-up estimates (and project schedule) have been developed and negotiated through this bottom-up/top-down process.

Numerous cost models have been developed over the past three decades. Perhaps the best known of these models are the COCOMO (COnstructive COst MOdel) family of models (Boehm and Turner, 2003) and Function Points (Albrecht, 1979). The original COCOMO model was a regression model that predicted a project’s cost based on a number of “factors”—project, process, hardware, and personnel attributes—that affected a project’s complexity and hence cost. Function points attempted to estimate a project’s size and complexity, and hence cost, by looking at the complexity of business requirements and processing requirements. While neither of these models is explicitly about architecture, their concerns and levels of abstraction were consistent with those of software architectural analyses.

1.4 Architecture-based benefit modeling

In the final analysis, the goal of economics-driven architecting is to create the maximum amount of value. Value should be our guide in making any architecture-related decisions. We need
value-based activities and strategies that are practical and easily implemented that have a principled basis and that have a simple and clear rationale.

What is value? This is, in itself, a complex issue when dealing with software. Value is how much a product or service is worth to a stakeholder, and this worth is often relative to other things. It is typically measured in money, but it does not always have to be. Value can also be an evaluation of what something could or should be worth, or an explanation of its actual market value (price); this is the purpose of using stock prices to value a company on the open stock market. Furthermore, we have to consider such facts as productivity, evolvability, and reputation, all of which are indirect ways in which a product or service might create value. These are frequently trickier to measure than “price” or “profit,” but they are extremely useful measures nonetheless! Finally, value in software is not just a function of costs such as personnel or material inputs; it reifies what the software is worth to a wide variety of stakeholders: the consumer, the producer, maintainer, and so on. For example, how valuable is it for the software architect to have an evolvable design?

When creating or evolving a software or system architecture, the architect needs to focus on value. Design is, in its essence, a search through a (virtually infinite) space of possible alternatives, choosing the one that maximizes expected value, given the project’s available resources, the schedule or deadline, and the set of project constraints (interfaces that must be complied with, resources that must be employed, Commercial Off-The-Shelf (COTS) packages that must be used, strategic agreements with partners, abilities of the development team, backward compatibility, etc.).

Shooting for the absolute “maximum” amount of value is not achievable in practice. Instead, in any value-based problem, we aim for something slightly less optimal but vastly more achievable: Pareto-optimality (also known as 80–20 rules). What is an 80–20 rule? In 1906 Vilfredo Pareto noted that 80% of income in Italy went to 20% of the population. He noted many such unequal distributions. For example, he reported that 20% of the pea plants in his garden produced 80% of the peas. He saw this as an optimization problem: How can one find the 20% in advance and concentrate one’s resources and efforts on this population?

Given a set of alternative allocations and a set of individuals, a movement from one allocation to another that can make at least one individual better off, without making any other individual worse off, is called a Pareto improvement. An allocation of resources is Pareto optimal when no further Pareto improvements can be made. How do we achieve this in practice? In software engineering such 80–20 rules have also been noted. For example, 80% of the defects are found in 20% of the code, or 80% of the value is found in 20% of the features. Of course, there is nothing magical about the numbers 80 and 20; it could be 90–10 or 60–40. The point is that we would like to be able to optimize our efforts and focus on the 20% of tasks that lead to 80% of the value. We would like to be able to predict where to spend our time. And this boils down to cost/benefit estimation and decision making.

One technique that is being adopted from the field of financial economics is use of the theory of real options to provide an estimate of value. An architectural decision, such as the application of a pattern, is analogous to a financial derivative. In financial markets a derivative is an instrument whose value depends on, or derives from, the values of basic underlying assets. For example, a stock option is a derivative whose value is dependent on the price of the stock. The stock is the underlying asset for the option.

Real options theory can be employed to analyze the value of architectural patterns in terms of the quality attributes, and hence value, that can be achieved with them (Ozkaya et al., 2007).
Options are valuable when there is uncertainty. The theory is typically applied in situations when uncertainty is large enough that it is necessary to wait for more information. Waiting can be valuable when it allows investments to be postponed until uncertainty is resolved. The components of real options analysis include the decision to make, characterization of the uncertainty, and the decision rule. The decision rule can be viewed as a simple mathematical expression that will serve as a guide on when to exercise the decision. The decision rule also helps identify the critical parameters that need to be observed in making the decision.

Clearly, the real options formulation is applicable to architectural design decisions: Investments are frequently contingent on prevailing and future market conditions (e.g., the addition of a new tier in an n-tier architecture makes it easier to make changes to business rules). In addition, there is typically a large degree of uncertainty in the future that makes it advisable to wait for more information before making a decision.

1.5 Architecture and risk management

Architectural design and analysis are intimately tied to risk management, which in turn supports project management. When making architectural decisions, the architect needs to consider the level of certainty of the requirements and how any consequent architectural decisions might change in the future. Each of these is, in essence, an estimate of risk.

Real options are one way of managing risk, by attempting to quantify the consequences of architectural decisions. Other forms of architectural risk management are creating performance models, building simulations of critical system behavior, and building architectural “experiments” to test out important new concepts or pieces of system infrastructure. These kinds of risk management typically focus on the runtime aspects of a system’s behavior.

But the non-runtime aspects can have an equally important impact on a system’s eventual success. Consider modifiability, for example. If the system is not “appropriately” modifiable, then it will likely fail. If, on the one hand, the system is hard to modify then it will not satisfy its user’s demands in the future and will fall behind competitive offerings. If, on the other hand, the system is designed to be extremely modifiable, it may take a very long time to reach the market, and it may be very expensive to create and highly complex to test.

The metaphor of “technical debt” (Brown et al., 2010) was created to describe the situation when developers make short-term suboptimal decisions that need to be “paid back” later in a project. The most common form of technical debt is related to modularity or, more precisely, the lack of modularity. By not modularizing functionality, developers can quickly develop code but typically at the expense of later modifiability. This debt accumulates over the life of a project, and it is only paid down by refactoring activities, which can be seen as a kind of investment in the existing software corpus. Numerous techniques have been suggested to attempt to locate and diagnose technical debt, including “code smells” (Fowler, 1999), modularity violation detection (Wong et al., 2011), clone detection (Kim Notkin, 2009), and various coupling and cohesion metrics (Chidamber and Kemerer, 1994). Each of these techniques is aimed at determining architectural “hot spots”—locations in the system where technical debt has accumulated.

Of course, other types of technical debt are possible as well, such as the extravagant use of resources in a first implementation, which then needs to be tuned and refined as a system matures.
and attempts to scale. Whatever the source and form of the technical debt, it is essentially an economic decision: Is it a choice of “how much architecture” and “how much risk mitigation” up-front versus getting a release to market more quickly, albeit with less-than-optimal quality attributes?

### 1.6 Architecture and agility

Agile processes were initially employed on small- to medium-sized projects with short time frames and enjoyed considerable early success. In the early years, Agile processes were not often used for larger projects, and Agile projects eschewed up-front planning, including architectural design.

Agile processes, when they first appeared, were a response to a need for projects to:

- be more responsive to their stakeholders.
- be quicker to develop functionality that users care about.
- show more and earlier progress in a project’s life cycle.
- be less burdened by documenting aspects of a project that would inevitably change.

These needs, however, are not in conflict with architecture. The question for a software project is not “Should I do Agile or architecture?” The Agile manifesto claimed to prefer individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan. Is any of this inimical to the use of architecture?

In fact, the question for a software project is not “should I do Agile or architecture?”, but rather, “how much architecture should I do up-front versus how much should I defer until the project’s requirements have solidified somewhat?”, or “when and how should I refactor?”, “how much of the architecture should I formally document, and when?” and “should I review my architecture and, if so, when?”

Boehm and Turner (2003), for example, have looked at the trade-offs between “agility” (the ability to quickly respond to changes) and discipline (up-front work on architecture and risk resolution). The more up-front work that you do, the more cost that you add, up-front, to a project. For small projects—say, 10,000 lines of code—this discipline simply never pays off. However, for medium (100,000 lines of code) and large (1,000,000 lines of code) projects, up-front work does pay off. In fact, the larger the project, the more that up-front work pays dividends. Every project has a sweet spot between agility and discipline. Bohem and Turner’s analysis helps to find this spot.

### 1.7 Runtime economics-driven architecting

The work on EDSA has also been concerned with dynamic and self-adaptive architecting for value creation. We describe research effort under this category as runtime EDSA. Runtime EDSA stems from the belief that the ability of a design decision to create value tends to be limited; it can be sensitive to dynamic changes in context, inputs from the environment, and users’ requirements. Research under this category has primarily been concerned with self-organization and self-optimization of the
architecture, with the objective of delivering an added value through optimal modes of architectural compositions, which are best “fit” to the runtime changes in requirements and the environment. In EDSA at runtime, value creation through optimization is formulated as a “dynamic search” problem. The search continuously evaluates and optimizes the structure, where value can be linked to behavior and context requirements, technical debt analysis and reduction, energy efficiency, risk mitigation for compliance, and improved Quality of Service (QoS). Design time decisions form a “portfolio” of possible strategies, which can be automatically searched and exercised at runtime if a strategy promises an added value. In this context, value creation is treated as a moving target, which can be self-optimized through feedback loops, measurement and control leading to self-organization, and self-management of the structure.

The application of runtime EDSA has been appealing to service-centric and cloud-based architectures in particular. Let us have a closer look at cloud-based architectures and how they can benefit from runtime EDSA by referring to (Nallur and Bahsoon, 2013; Faniyi Bahsoon, 2012). Given the unpredictable, highly dynamic, elastic and on-demand nature of the cloud, priory knowledge and design time strategies for value creation are mere difficult to formulate. It would be unrealistic to assume that optimal strategies for value creation can be predicted at design time as value is heavily dependent on dynamics in the infrastructure inducing these architectures.

Cloud-based architectures can be composed of web services, which can be leased off the cloud. The value of the application and its underlying architecture is a function of the value of the individual web services composing the application. These architectures can “trade” instances of abstract web services off the cloud, which can serve as a basis for dynamic synthesis and composition of the architecture. That is, for a given abstract service A, there exist multiple concrete services $A_1, \ldots , A_n$ in the marketplace offering comparable functionalities but differing in their price and QoS provisions. We view the cloud as a marketplace. Cloud-based architectures encompass a set of strategies and tactics that can be viewed as goods to be traded in this market. We argue that market-based control can contribute to the foundation of the runtime EDSA. Economics and markets have intrinsic properties that make them of interest in this setting, namely, decentralization, scalability, adaptiveness, and robustness. Market mechanisms lend themselves neatly to the notion of valuation of cost and benefits via utilities. “Buyers” are cloud-based architectures, and sellers are vendors and providers of the services, which realize quality attributes, functional requirements, and price and environmental constraints (Faniyi Bahsoon, 2012).

In a dynamic and continuously evolving context, the challenge is to understand how value can be captured, modeled, and analyzed and to discover the various trade-offs involved and their evolution trends as the system is executing. A common way of representing value for an architecture tactic (e.g., provision of resource to improve QoS) is to formulate the value using utility functions. This effort could take different forms (e.g., additive utility), where each formulation impacts the market’s ability to reach desirable emergent state(s). There are many existing market mechanisms in the micro-economics literature. Each theoretically studied mechanism can be shown to reach some known solution concept (e.g., Nash equilibria). A self-adaptive mechanism, which uses these utility models, can assist analysts, architects, and enterprises in providing built-in support for an autonomic architecture management supporting continuous evolution for value added. The inputs to these modes can be continuous data and feeds available via published benchmarks (e.g., cloudharmony.com; spec.org), which can continuously steer the selection and composition of such architectures.
1.8 Final thoughts

A book such as this can only be a snapshot of a complex and growing field. We have attempted to survey the major historical and current research areas within economics-driven software architecture, but such a picture will always be incomplete and subject to revision. Such is life.

References


