Chapter 17. Orchestration-Driven Service-Oriented Architecture

Architecture styles make sense to architects in the context of the era when they evolved, but lose relevance in later eras, much like art movements. *Orchestration-driven service-oriented architecture* (SOA) exemplifies this tendency. The external forces that often influence architecture decisions, combined with a logical but ultimately disastrous organizational philosophy, doomed this architecture to irrelevance. However, it provides a great example of how a particular organizational idea can make logical sense yet hinder the most important parts of the development process. It illustrates one of the dangers of ignoring our First Law: *everything in software architecture is a trade-off*.

Topology

The topology of orchestration-driven SOA is shown in Figure 17-1.

Not all examples of this style of architecture have these exact layers, but they all follow the same idea of establishing a taxonomy of services within the architecture, each layer with a specific, well-defined responsibility.

Service-oriented architecture is a distributed architecture. The exact demarcation of boundaries isn't shown in Figure 17-1 because it varies based on organization and tools: some of the parts of the taxonomy may exist inside an application server. Orchestration-driven SOA centers on a specific taxonomy of services, with different technical responsibilities within the architecture and roles dedicated to specific layers.

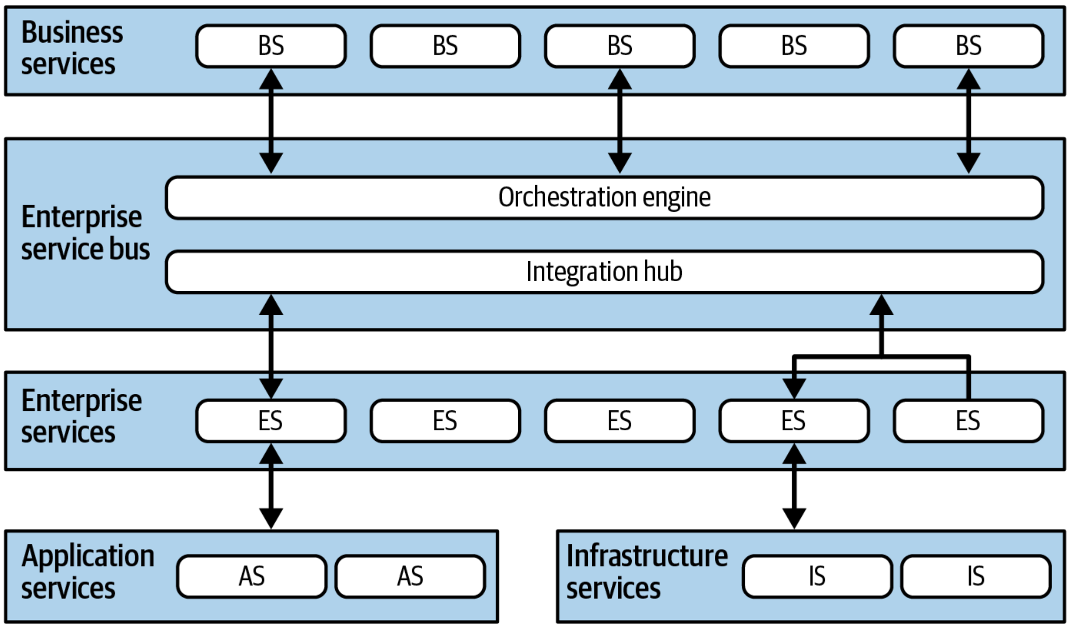


Figure 17-1. Topology of orchestration-driven service-oriented architecture

Style Specifics

Orchestration-driven SOA is mostly of historical interest to current-day architects; the lessons learned from building these architectures are an important part of the field's evolution. However, architects still find parts of it relevant in some integration architecture scenarios.

Service-oriented architecture appeared in the late 1990s, just as small companies of all kinds were growing into enterprises at a breakneck pace, merging with smaller companies, and requiring more sophisticated IT to accommodate this growth. However, computing resources were scarce, precious, and commercial. Distributed computing had just become possible and necessary, and many companies needed its scalability and other beneficial characteristics.

Many external drivers forced architects in this era toward distributed architectures with significant system constraints. Before open source operating systems were thought reliable enough for serious work, operating systems were expensive and licensed per machine. Similarly, commercial database servers came with Byzantine licensing schemes, which sometimes caused application-server vendors (which offered database connection pooling) to react by battling with database vendors. Given that so many resources were expensive at scale, architects adopted a common philosophy of reusing as much as possible.

These technical concerns wedded with organizational concerns about duplication of both information and workflows—because of frequent mergers and growth, organizations struggled with the variety and inconsistencies between core business entities. This made the goals of SOA attractive. Consequently, architects embraced *reuse* in all forms as the dominant philosophy in this architecture, the side effects of which we cover in "Reuse…and Coupling". This style of architecture also exemplifies how far architects can push the idea of technical partitioning, which can have good motivations but leads to bad consequences when taken to extremes.

Why So Many Service Names?

Architects are often confused by the multitude of things in software architecture called *services*. We cover three different "services" styles even in this book: SOA in this chapter, microservices in Chapter 18, and service-based architectures in Chapter 14. Part of the problem is the inflexibility of language to describe architectures. The other part lies in the constant evolution of the software-development ecosystem. *Service* is a nice generic name for something that provides a, well, service, so architects tend to reuse the name. As styles evolve, we change what we mean by *service*. For example, an *entity service* in an orchestration-driven SOA is different in virtually every way from a service in a microservices architecture (which is distinct from a service-based architecture). As annoying as it is, architects must often parse the context when the word *service* appears in a name; the term itself has suffered from semantic diffusion.

Taxonomy

The driving philosophy behind this architecture is a specific type of abstraction and enterprise-level reuse. Many large companies were annoyed at how much they had to continually rewrite software. They arrived at a strategy that seemed to solve that problem gradually by creating a strict *service taxonomy*, with well-defined layers and corresponding responsibilities. Each layer of the taxonomy supports the dual goals of ultimate abstraction and reuse.

Business services

Business services sit at the top of this SOA and provide the entry point for business processes. For example, services like `ExecuteTrade` or `PlaceOrder` represent the correct scope of behavior for these services. One litmus test common at the time was: can an architect answer “yes” to the question “Are we in the business of…” for each of these services? If so, then the service is at the appropriate level of granularity. However, while a developer might need a method like `CreateCustomer` to carry out a business process like `ExecuteTrade`, `CreateCustomer` is at the wrong level of abstraction for a business service. The company isn’t in the *business* of creating customers, but it needs to create customers in order to execute trades.

These service definitions contain no code—just input, output, and sometimes schema information. Business users and/or analysts define these service signatures, hence the name *business services*.

Enterprise services

The *enterprise services* contain fine-grained shared implementations. Typically, a team of developers builds atomic behavior around particular business domains, such as `CreateCustomer` or `CalculateQuote`, and transactional entities, such as `Customer`, `Order`, and `Lineitem`. These enterprise services are the building blocks that make up the business services, tied together via the orchestration engine.

It is worth noting the abstraction differences between business and enterprise services. While business services are quite coarse-grained, enterprise services are fine-grained and meant to capture different types of abstractions, workflows, and entities. The architect’s goal in creating enterprise services is to create perfectly encapsulated building blocks of isolated business functionality that can be freely composed into more complex business workflows.

While that’s a laudable goal, architects find that the ideal sweet spot of abstraction between all these forces is elusive at best and likely impossible because of numerous competing trade-offs. Ultimately, like other technically partitioned architectures, this architecture attempts a strict separation of responsibility, which flows from the imperative of reuse. The idea is that if developers can build fine-grained enterprise services at just the correct level of granularity, the business won’t have to rewrite that part of the business workflow again. Gradually, the business will build up a collection of reusable assets in the form of reusable enterprise services—in theory, anyway.

Unfortunately, the dynamic nature of reality and the evolutionary impact of the software development ecosystem defy these attempts. Business components aren’t like construction materials, where solutions last decades. Markets, technology changes, engineering practices, and a host of other factors confound attempts to impose stability on the software world.

Application services

Not all services in the architecture require the same level of granularity or reuse as the enterprise services. *Application services* are one-off, single-implementation services. For example, perhaps one application needs geolocation, but the organization doesn’t want to take the time or effort to make that a reusable service. An application service, typically owned by a single application team, solves that problem.

Infrastructure services

Infrastructure services supply operational concerns, such as monitoring, logging, authentication, authorization, and so on. These services tend to be concrete implementations, owned by a shared infrastructure team that works closely with operations. Architects' philosophy in building this architecture revolves around technical partitioning, so it makes sense that they would build separate infrastructure services.

Orchestration engine and message bus

The *orchestration engine* forms the heart of this distributed architecture, stitching together the business service implementations using orchestration, including features like transactional coordination and message transformation. The orchestration engine defines the relationship between the business and enterprise services, how they map together, and where transaction boundaries lie. It also acts as an integration hub, allowing architects to integrate custom code with package and legacy software systems. This combination of features highlights the modern-day use of tools like enterprise service buses (ESBs). While most architects consider it a bad idea to build an entire architecture around ESBs, they are immensely useful in integration-heavy environments. Where architects must combine an integration hub and orchestration engine, why not use a tool that already includes them? (This points to another important skill to develop as an architect—how to discern the true uses of tools, separate from the hype, both good and bad.)

Because the message bus forms the heart of the architecture, Conway's Law (see "Conway's Law") correctly predicts that the team of integration architects responsible for this engine tends to become a political force within an organization —and eventually a bureaucratic bottleneck.

While this centralized, taxonomized approach might sound appealing, in practice it has mostly been a disaster. Offloading transaction behavior to an orchestration tool sounds good, but architects struggle to find the correct level of granularity. While they can build a few services wrapped in a distributed transaction, the architecture becomes increasingly complex. Developers must figure out where the appropriate transaction boundaries between services lie as entities become involved in numerous workflows. Despite managers predicting and hoping that organizations could successfully build transactional building blocks as enterprise services, it has proved difficult in practice.

Message flow

All requests go through the orchestration engine, where this architecture's logic resides. Thus, message flow goes through the engine even for internal calls, as shown in Figure 17-2.

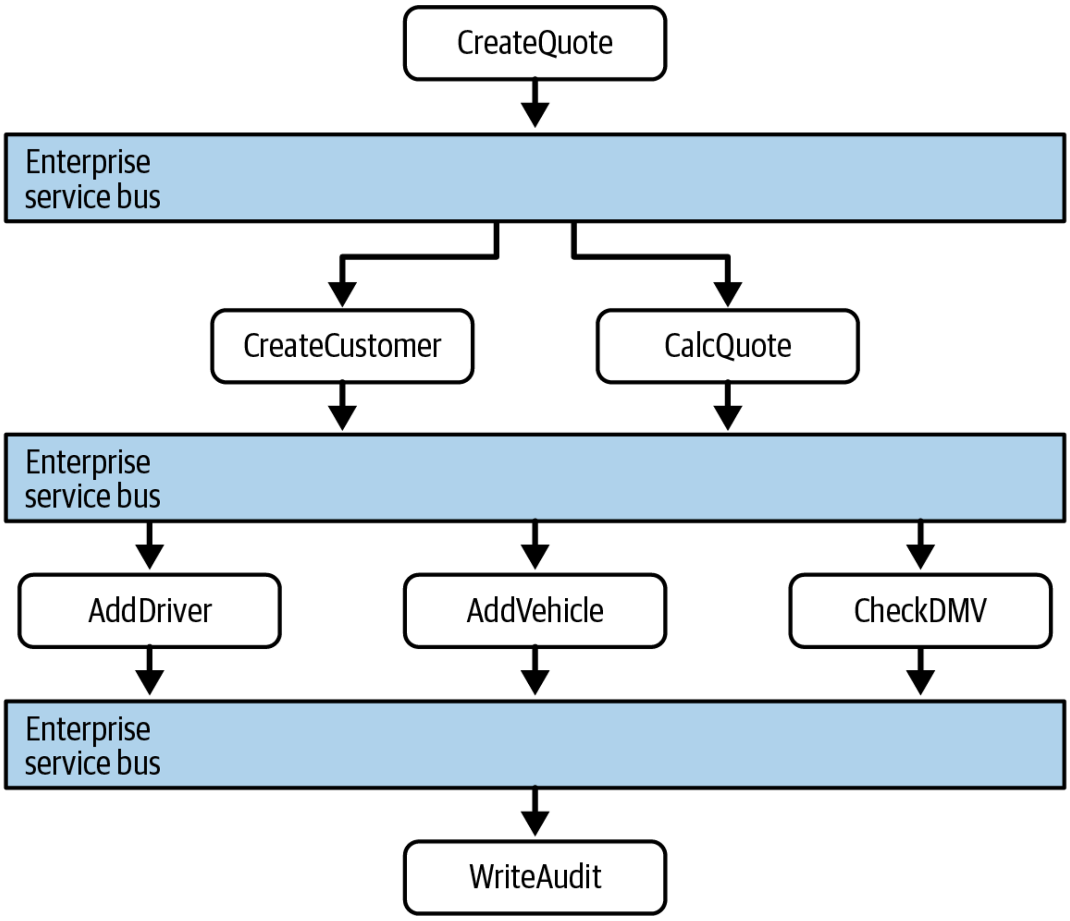


Figure 17-2. Message flow with service-oriented architecture

In Figure 17-2, the `CreateQuote` business-level service calls the service bus, which defines the workflow. The workflow consists of calls to both `CreateCustomer` and `CalculateQuote`, each of which also makes calls to application services. The service bus acts as the intermediary for all calls within this architecture, serving as both an integration hub and orchestration engine.

Reuse…and Coupling

One of the primary goals of the architects who first utilized this architecture was reuse at the service level—the ability to gradually build business behavior that they could incrementally reuse over time. They were instructed to find reuse opportunities as aggressively as possible.

For example, consider the situation illustrated in Figure 17-3. An architect realizes that each of six divisions within an insurance company contains a notion of `Customer`.

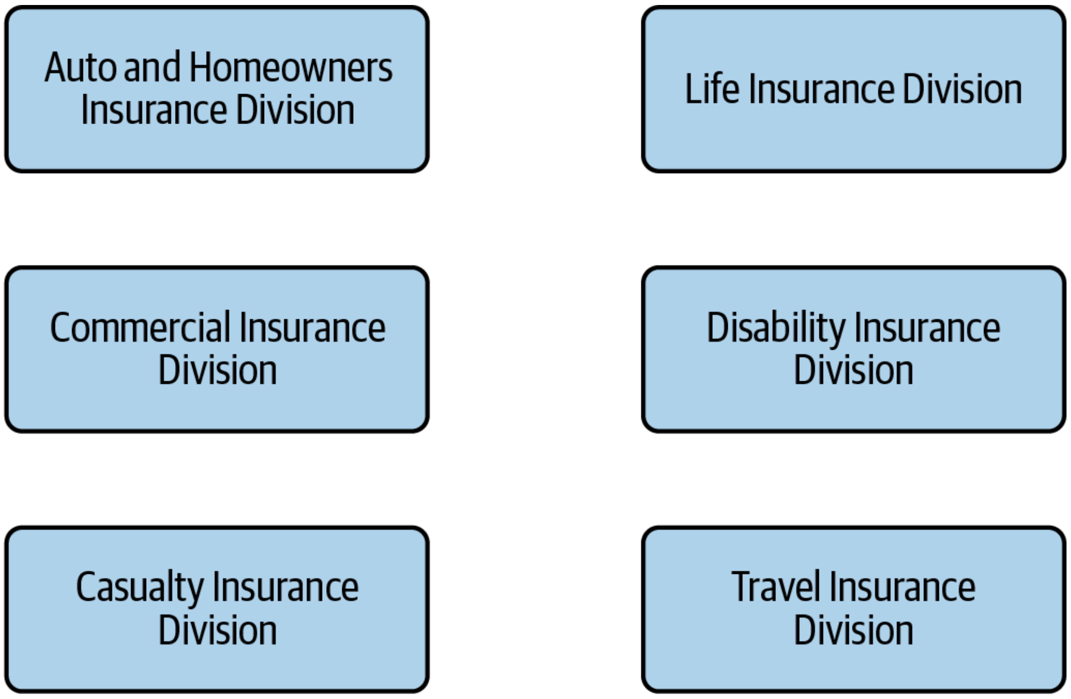

Auto and Homeowners Insurance Division
Life Insurance Division
Commercial Insurance Division
Disability Insurance Division
Casualty Insurance Division
Travel Insurance Division

Figure 17-3. Seeking reuse opportunities in service-oriented architecture

Therefore, the proper SOA strategy entails extracting the `Customer` parts into a reusable service and then allowing the original services to reference the canonical `Customer` service. This is shown in Figure 17-4: here the architect has isolated all customer behavior into a single `Customer` service, achieving the obvious reuse goals.

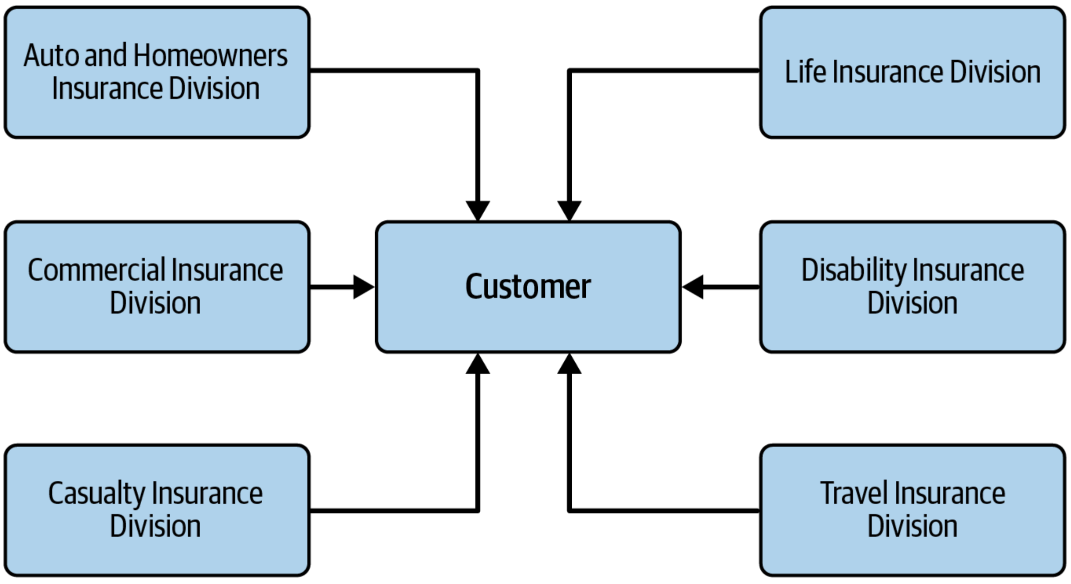

Auto and Homeowners Insurance Division
Life Insurance Division
Commercial Insurance Division
Customer
Disability Insurance Division
Casualty Insurance Division
Travel Insurance Division

Figure 17-4. Building canonical representations in service-oriented architecture

Architects only slowly realized the negative trade-offs of this design. First, when a team builds a system primarily around reuse, it also incurs a huge amount of coupling between components. After all, reuse is implemented *via* coupling. For example, in Figure 17-4, a change to the `Customer` service ripples out to all the other services. This makes even incremental change risky: each change has a potentially huge ripple effect. That in turn requires coordinated deployments, holistic testing, and other drags on engineering efficiency.

Another negative side effect of consolidating behavior into a single place: consider the case of auto and disability insurance in Figure 17-4. To support a single `Customer` service, each division must include all the details the organization knows about its customers. Auto insurance requires a driver's license, which is a property of the person, not the vehicle. Therefore, the `Customer` service will have to include details about driver's licenses that the disability insurance division cares nothing about. Yet the disability insurance team must deal with the extra complexity of a single customer definition. In many ways, DDD's insistence on *avoiding* holistic reuse derives from experiences with these kinds of architectures.

Perhaps the most damaging revelation about orchestration-driven SOA is the impracticality of building an architecture so focused on technical partitioning. While this makes sense from a separation and reuse philosophy standpoint, it is a practical nightmare.

For example, developers commonly work on tasks like "add a new address line to `CatalogCheckout`." Domain concepts like `CatalogCheckout` are spread so thinly throughout this architecture that they were virtually ground to dust. In an SOA, this task could involve dozens of services in several different tiers, plus changes to a single database schema. What's more, if the current enterprise services aren't defined at the correct transactional granularity, the developers will either have to change their design or build a nearly identical new service to change the transactional behavior. So much for reuse.

Data Topologies

Unlike many of the architecture styles we discuss in this book, the data topologies for orchestration-driven SOA aren't very interesting, given this style's historical origins. Even though it is a distributed architecture consisting of many parts, it generally uses a single (or a few) relational databases, which was the common practice in all distributed architectures in the late 1990s. Even transactionality was generally relegated to this architecture and away from databases: the message bus often included declarative transactional interactions for each of the entities within the topology, allowing developers, architects, or others to determine transactional behavior independently of the database or even the situational reuse of the entity.

Architects in this era considered data a foreign country. While it is an inevitable part of the plumbing in both SOA and event-driven architectures, back then, they treated it more as an integration point than as part of the problem domain.

Really? Declarative Transactions?!?

Yes, really. One of the "features" of many application servers in the heyday of orchestration-driven SOA was allowing configuration managers to change the transactional scope of individual entities, depending on the transactional context within which they wanted to operate. (This was declared, of course, in XML, given the era and its love for verbose but easy-to-parse configuration formats.) A portion of the declaration of an entity (called `EntityBeans`, a specialized type of JavaBean) determines transactional scope as it participates in workflows, which architects themselves declare to be transactional or not. In turn, the application server interacts with the database to create and manage database transactions that match the desired behavior of the entities and/or workflows.

This has largely failed, for two reasons. First, if a developer doesn't know what the transactional behavior will be at runtime, it adds considerable complexity to entities and dependencies. This forces developers to create almost identical versions of the entities, differing only in their transactional scope. Second, no matter how much sophistication vendors build into their message buses, edge cases constantly appear where the myriad failure modes prevent the system from cleanly managing transactions, creating tangled messes of inconsistency for humans to untangle. Some complex, multifaceted features of systems (like transactions) cannot be cleanly abstracted away. Too many leaks in the abstraction prevent it from achieving reliability.

Cloud Considerations

Orchestration-driven SOA predates the cloud by several decades, so no consideration exists for building this architecture (in its original incarnation) in the cloud.

However, the current-day use of this style makes it a good integration architecture for cloud and on-premises services that must integrate and participate in workflows. As primarily an integration architecture, it works well with cloud-based services and facilities.

Common Risks

At the end of the last century and the beginning of this one, the big risks for this architecture were primarily about how much it cost, how long it took to implement, and (a shocking surprise) how difficult these systems are to maintain and update. Many of these projects were very expensive multiyear endeavors, with critical decisions made high in the company hierarchy. Rather than call these projects “failures,” companies mostly just transformed them into integration architectures with better boundaries, more closely aligned with the ideas of DDD.

When architects use an ESB in a modern system as an integration facility, the biggest risk is the slippery slope of allowing the ESB to gradually encapsulate the entire architecture. This is called the *Accidental SOA* antipattern: where an architect gradually and unintentionally builds a fully orchestration-driven SOA without realizing it. To avoid Accidental SOA, an architect must ensure reasonable encapsulation boundaries for orchestration and pay close attention to issues like transactional boundaries.

Governance

When this architecture was popular, modern-day holistic testing was uncommon. Teams rarely tested SOA outside of formal quality-assurance-level testing, so tool and framework creators gave little consideration to facilitating testing the individual parts. Some testing frameworks emerged to create mocks and stubs for the massive machinery of the message bus and its related moving parts, but they were always cumbersome and inconsistent.

Governance suffered from the same limitations. The idea of automating architectural governance was even more foreign than automating testing. In this era, “governance” meant heavyweight frameworks, meetings, and code reviews—all manual.

However, architects still use ESBs strategically within organizations that need the particular mix of capabilities they offer. In particular, many companies have legacy systems that must interact with more modern systems, often combining results and aggregating behavior—all of which describes the central functionality of an ESB. In these scenarios, fitness functions can serve a critical role in preventing data or bounded contexts from “leaking” across parts of the ecosystem where they shouldn’t appear.

For example, consider a system that uses an ESB to coordinate between an enterprise resource planning (ERP) package, an online sales tool, and more modern microservices-based `Accounting` services. In this scenario, the system should only read from the ERP and sales systems and should write to the `Accounting` microservices. Architects can first build a fitness function that ensures that all communication is written consistently to logs, then write something like the following fitness function (given here in pseudocode):

```
READ logs for ERP into ERP-logs for past 24 hours
READ logs for Sales into Sales-logs for past 24 hours
FOREACH entry IN ERP-logs
    IF 'operation' is 'update' and 'target' != 'accounting' THEN
       raise fitness function violation
             "Invalid communication between integration points"
    END IF
FOREACH entry IN Sales-logs
    IF 'operation' is 'update' and 'target' != 'accounting' THEN
       raise fitness function violation
             "Invalid communication between integration points"
    END IF
```

This fitness function reads the log entries from both integration points to ensure that no update operations take place whose target isn’t the `Accounting` system.

Using such fitness functions, architects can use tools like ESBs strategically while building guardrails around the places where teams typically misuse them.

Team Topology Considerations

Just as architects did not consider data topologies for orchestration-driven SOA, the same is true for team topologies, which was an unknown topic when this architecture style was popular.

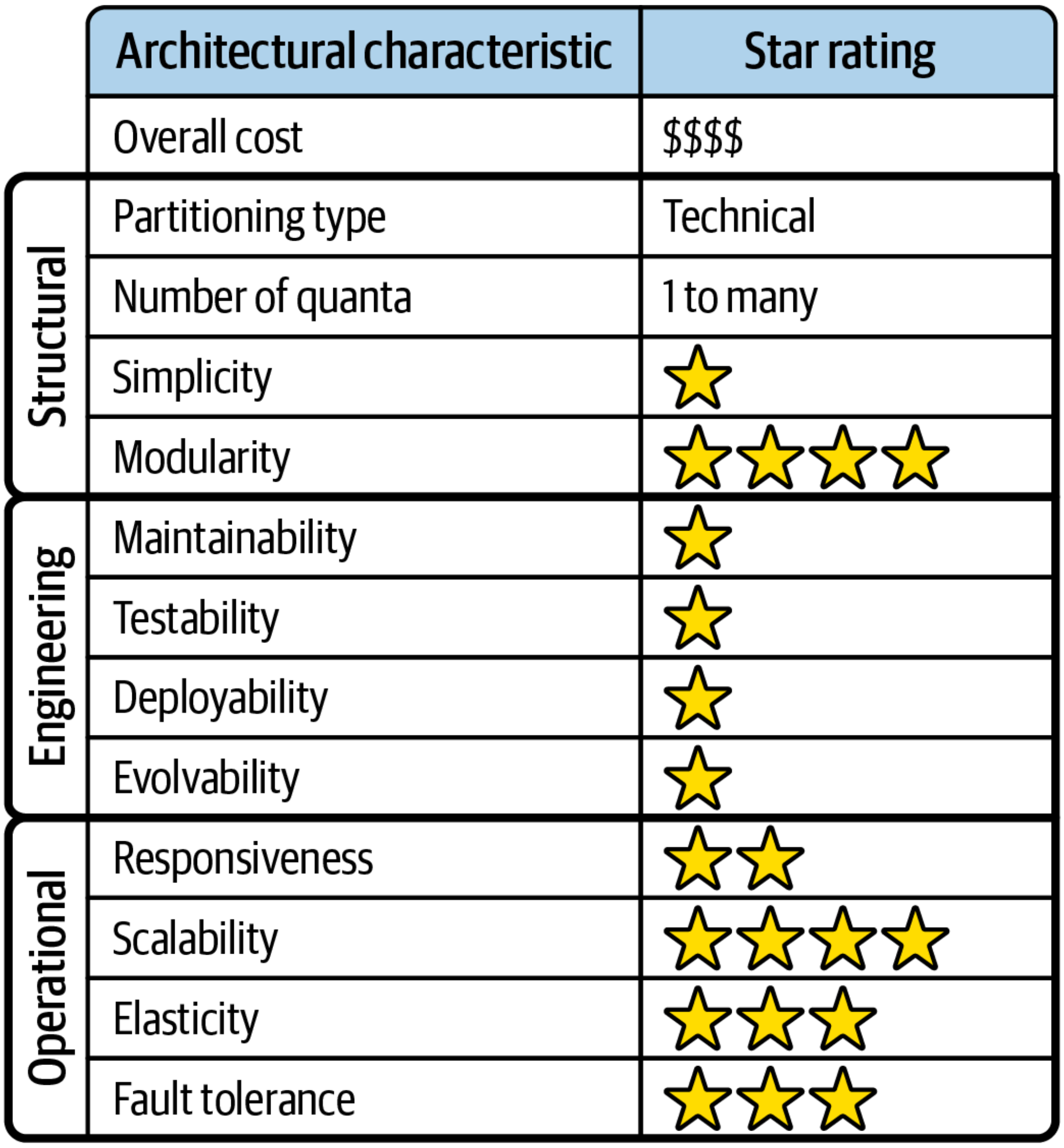
In fact, the strict taxonomy of this style serves as a communication antipattern that *led* architects to develop the principles of team topologies. The *goal* in this architecture is extreme separation of responsibilities, with a corresponding separation of team members. Among the companies that adopted it, it was rare indeed for someone building *business services* to chat with someone building *enterprise services*. They were expected to communicate through technical artifacts, such as contracts and interfaces. The level of abstraction in this style creates many integration layers, each implemented by different teams, using enterprise-level ticketing tools to communicate. It should be easy to see why developers find it time-consuming to build features in this style.

Style Characteristics

Many of the criteria we now use to evaluate architecture styles were not priorities when orchestration-driven SOA was popular. The Agile software movement had just started and had not yet penetrated the large organizations likely to use this architecture.

A one-star rating in the characteristics ratings table in Figure 17-5 means the specific architecture characteristic isn’t well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the style’s strongest features. Definitions for the characteristics identified in the scorecard can be found in Chapter 4.

SOA is perhaps the most technically partitioned general-purpose architecture ever attempted! In fact, the backlash against the disadvantages of this structure led to more modern architectures, such as microservices. SOA has a single quantum, even though it is a distributed architecture, for two reasons. First, it generally uses a single database or just a few, creating coupling points across many different concerns. Second, and more importantly, the orchestration engine acts as a giant coupling point—no part of the architecture can have different characteristics than the mediator that orchestrates all behavior. Thus, this architecture manages to find the disadvantages of both monolithic *and* distributed architectures.



	Architectural characteristic	Star rating
	Overall cost	$$$$
Structural	Partitioning type	Technical
	Number of quanta	1 to many
	Simplicity	★
	Modularity	★★★★
Engineering	Maintainability	★
	Testability	★
	Deployability	★
	Evolvability	★
Operational	Responsiveness	★★
	Scalability	★★★★
	Elasticity	★★★
	Fault tolerance	★★★

Figure 17-5. Service-oriented architecture characteristics ratings

Modern engineering goals such as deployability and testability score disastrously in this architecture, both because they are poorly supported and because those were not important (or even aspirational) goals when it was developed.

This architecture does support some goals, such as elasticity and scalability, despite the difficulties in implementing them. Tool vendors poured enormous effort into making these systems scalable by building session replication across application servers and other techniques. However, this being a distributed architecture, performance has never been a highlight, because each business request is split across so much of the architecture.

Because of all these factors, simplicity and cost have the inverse of the relationship most architects would prefer. Orchestration-driven SOA was an important milestone because it taught architects the practical limits of technical partitioning and how difficult distributed transactions can be in the real world.

Examples and Use Cases

The primary examples of this architecture existed in the late 1990s and early 2000s in many large enterprises. They have gradually been displaced by more agile and domain-based distributed architectures, like microservices. Even large enterprises have realized that change is inevitable and that software isn't static and needs to change with market forces and new capabilities.

Architects built orchestration-driven SOA architectures to try to achieve effective reuse across large organizations, but eventually realized how difficult their strict and elaborate taxonomy makes implementing common changes and updates. For example, a common domain change might be to update details about a single entity. On a lucky day, developers might only need to change components in the enterprise services layer to do the job. However, on a bad day (one where enterprise architects and/or business stakeholders didn't anticipate this type of change), developers might have to update four or five layers in the architecture, making highly coupled changes in each. Architects working in this style dread hearing the word *change* because it requires deep analysis, and the scope of the work is so variable.

As we mentioned in "Governance", architects still use the building blocks of orchestration-driven SOA (such as the ESB), particularly for integration architectures. For example, an ESB includes both an integration hub (to facilitate communication, protocol, and contract transformation) and an orchestration engine (to allow architects to build workflows between various integration endpoints). Because the orchestration-driven SOA includes many layers of indirection, it allows architects to implement enterprise services as integration points, as package software, or as bespoke code, among other possibilities, as illustrated in Figure 17-6.

Client requests use the message bus to determine which enterprise services to call, in what order to call them, and what information to aggregate. The enterprise services, in turn, communicate via APIs to custom code, old systems, or package software, and so on.

Orchestration-driven SOA represents an interesting innovation for how architects handle problems of integration at scale within the constraints of their ecosystem. For example, since most organizations weren't using open source operating systems when this architecture was popular, alternative architectures like microservices were impossibly expensive. Architects should learn from past approaches. We can continue using the parts that still make sense, while internalizing the lessons of what failed and why.

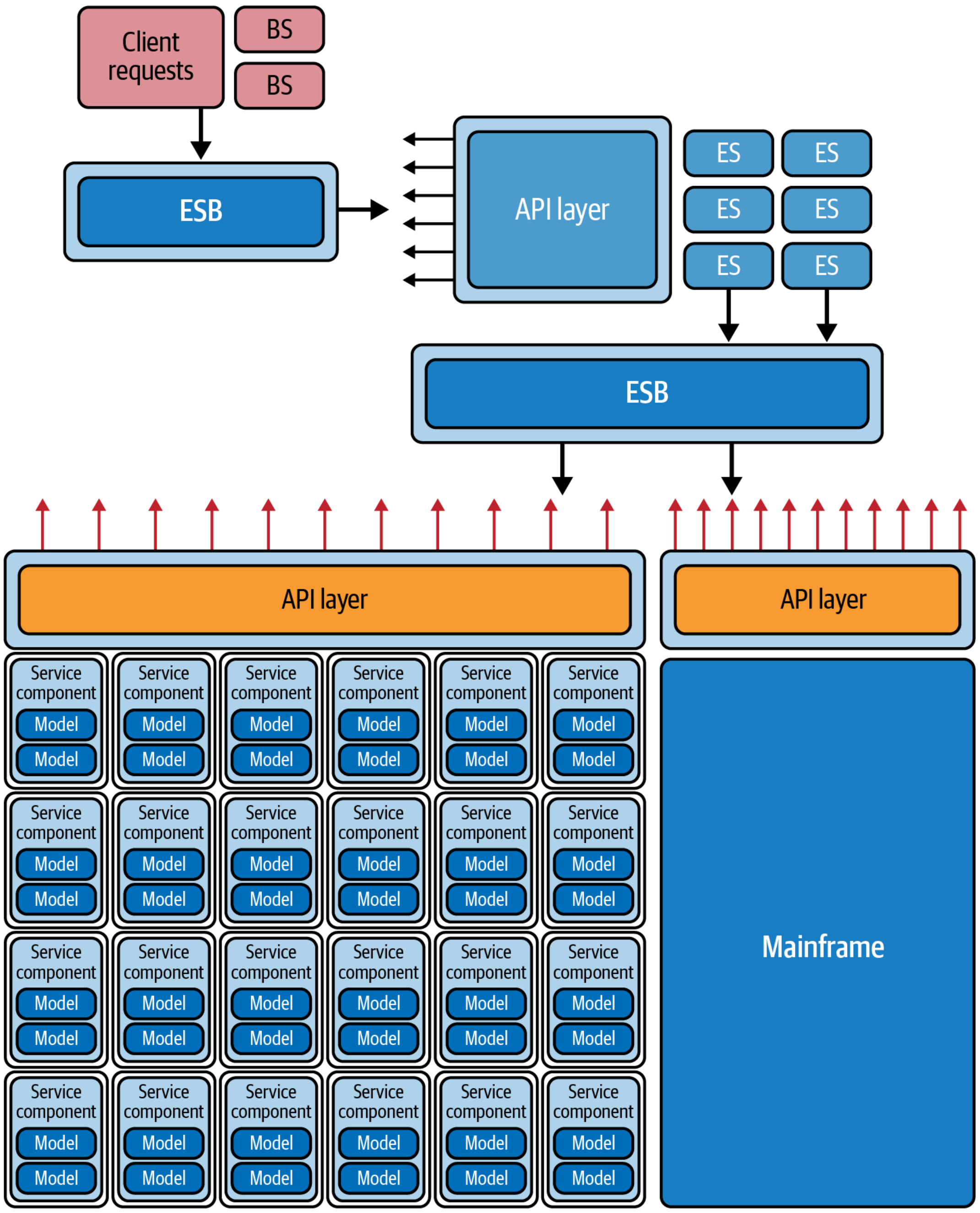


Figure 17-6. The layers of abstraction in this architecture style allow for implementation flexibility