Fundamentals of Software Architecture

An Essential Guide to Modern Software Design and Development Practices

---

Outline

Setting Expectations

The Big Picture : Understanding the Essence and Imperative of Software Architecture Design

- The Pivotal Role of Software Architecture in the Software Development Life Cycle (SDLC)
- Define Software Architecture
- Why need a good software architecture
- Software Architecture Scope
    - Characteristics
    - Structure
    - Behavior
    - Tech Stack

How To Decide and Create the Software Architecture

- Architecture Thinking & Planning
    - Architecture vs Design
    - Technical Breadth and Depth
    - Analyzing Trade-offs
    - Understanding the Business Drivers
    - Balancing Architecture and Hands-on Coding
- Architecture Principles

    - SoC & Modularity & SRP Key Differences Introduction
    - SoC
    - Modularity
        - Cohesion
        - Coupling
    - Bounded Context
- Architecture Characteristics
    - Characteristic Meets Criteria
    - Different Categories and Types of Characteristics
    - Identifying Architectural Characteristics
    - Governance and Fitness Functions
    - Scope of Architecture Characteristics
- Architecture Structure
    - Component-based thinking
    - Key aspects of defining architecture style
    - Architecture Styles/Patterns
        - Fundamental Pattern: Client/Server
        - Monolithic Versus Distributed Architectures
    - Microservices Architecture
    - Choosing the Appropriate Architecture Style
        - Decision Criteria
            - Team Structure Assessment
        - Case Study of Migration from Monolith to Microservice Architecture for an Industrial Enterprise.
- Technologies Stack Choice
    - Team Dynamics and Skill Assessment

Analyzing Architecture Risk

- Why Important ?
- Risk Storming
- Agile Story Risk Analysis

Documenting Architecture Decisions

- Why Do We Need Documentation?
- Various Documentation Types Needed based on the scope of the architecture specified.

This is a field that I found both confusing and intimidating when I was first starting my journey into software engineering.

So I wrote this to ensure that you don't have to go through the same confusion!

Setting Expectations

Key Focus Areas for a Comprehensive Exploration:

- **The Big Picture and Software Architecture:**
  This quick look takes us through What is software architecture and how it plays a key part in different ways of building software and its crucial role in the “BIG PICTURE” software development life cycle (SDLC).
- **In-Depth Exploration:**
  We'll get into the basics of what makes up software architecture, including its characteristics, principles, styles, design patterns, and how these elements work together to clear up any confusion.
- **Robust Design, Minus Overengineering:**
  Architecture Principles, Solid Principles, emphasizing the avoidance of overcomplication.
- **Architecture Planning and Decision Making:**
  Architecture Thinking, discusses strategic planning in different methodologies to make the right long term architectural decisions that fits the business needs and affect positively on the development process and the stakeholders involved.
- **Domain Driven Design,Your Secret Weapon:** We discuss the usage of Domain Driven Design as crucial design principles and practices for complex business implementation that leads to efficient system design.
- **Microservice Migration Case Study:** Blends theory with practice to illustrate the impact of architectural decisions.

The Value of This Exploration:

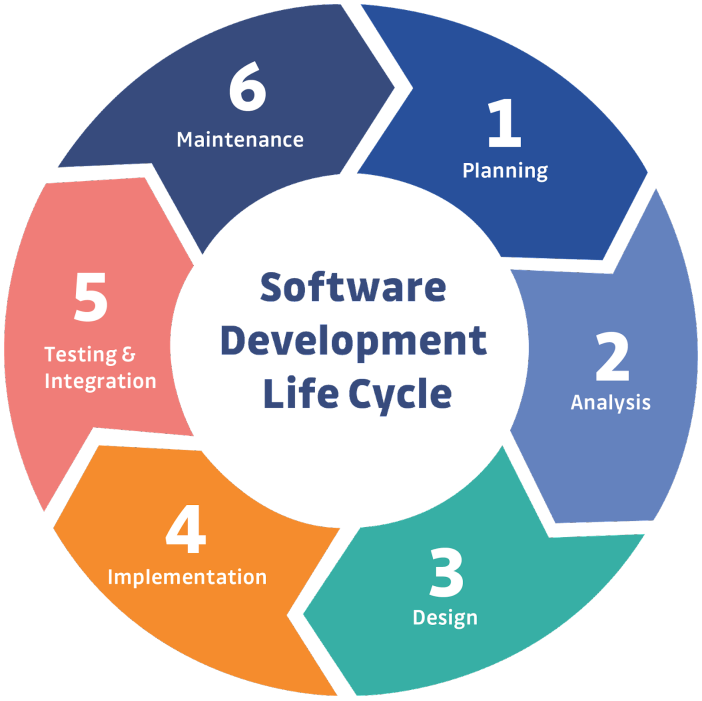
This journey through software architecture aims to create systems that are functional, adaptable, , and future-proof, meeting client and business needs without overengineering. Understanding software architecture's role in the SDLC is key to making informed decisions that result in effective, maintainable software solutions.

For each topic I'll give a brief and theoretical introduction. Then I'll share some examples to give you a clearer idea of how these work. Let's get to it!

Introduction

The Big Picture : Understanding the Essence and Imperative of Software Architecture Design

The Pivotal Role of Software Architecture Design in the Software Development Life Cycle (SDLC)



After gathering of business requirements by a business analyst then the developer team starts working on the , sequentially it undergoes various steps like testing, acceptance, deployment, maintenance etc. Every software development process is carried out by following some sequential steps which come under this Software Development Life Cycle (SDLC).

In the design phase of Software Development Life Cycle the software architecture is defined and documented. So in this article we will clearly discuss one of the significant elements of the Software Development Life Cycle (SDLC) i.e. the Software Architecture.

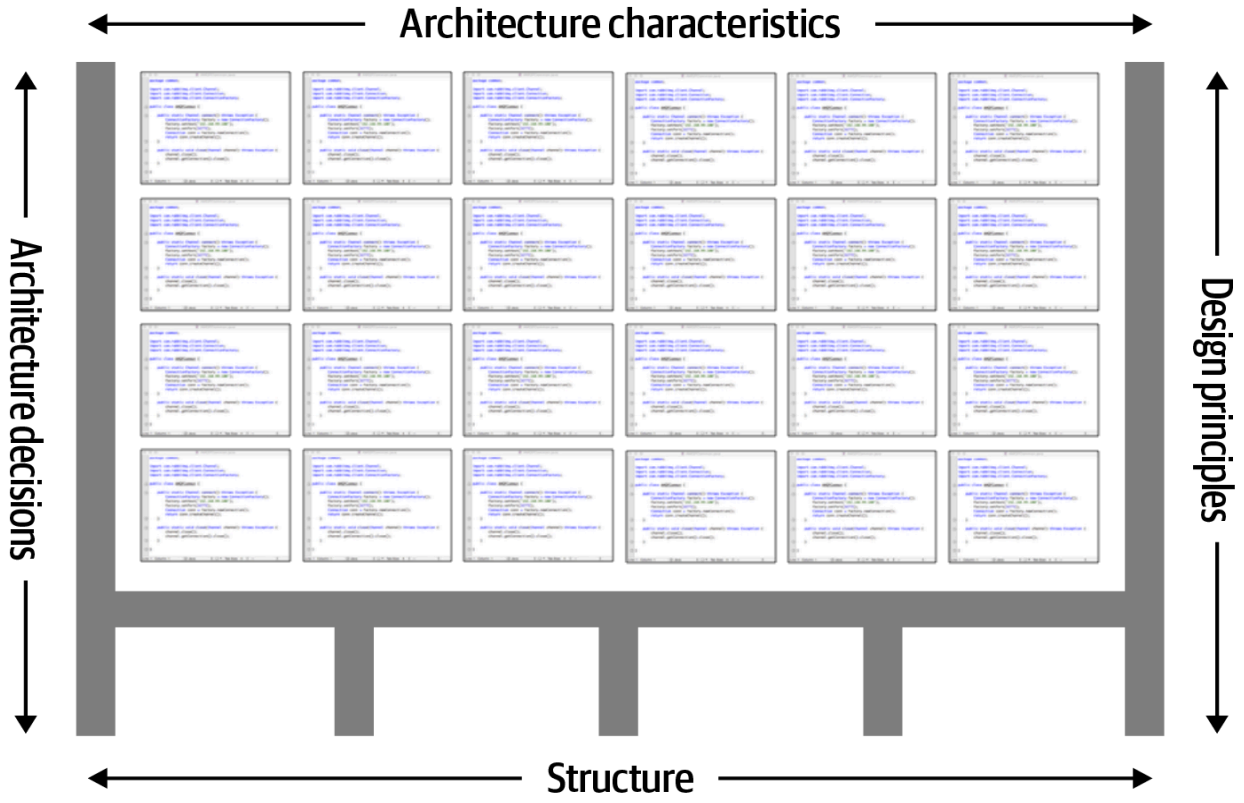
- **In Agile Methodologies**: Emphasizes flexible, evolving architecture to accommodate iterative development and changing requirements.
- In Agile, **architectural decisions** are made continuously, adapting to changing requirements and feedback. The emphasis is on making decisions that are good enough for now and safe enough to try, which can later be iteratively refined.
- **In the Waterfall Model**: Focuses on a more static, predefined architecture that guides the sequential development process.
- In Waterfall, **architectural decisions** made during the design phase have a long-lasting impact. As such, there's a greater emphasis on getting these decisions right the first time, as changes later in the process can lead to significant rework.

- **Software architecture** has significant relevance to **DevOps and CI/CD**. The design of software architecture can greatly influence the ease and effectiveness of implementing continuous integration (CI) and continuous deployment (CD) practices. **A well-designed architecture** that supports modularity, scalability, and flexibility can facilitate smoother and more efficient CI/CD processes by enabling faster integration of changes, more reliable automated testing, and more straightforward deployment strategies. Thus, thoughtful consideration of software architecture is essential for maximizing the benefits of DevOps practices.

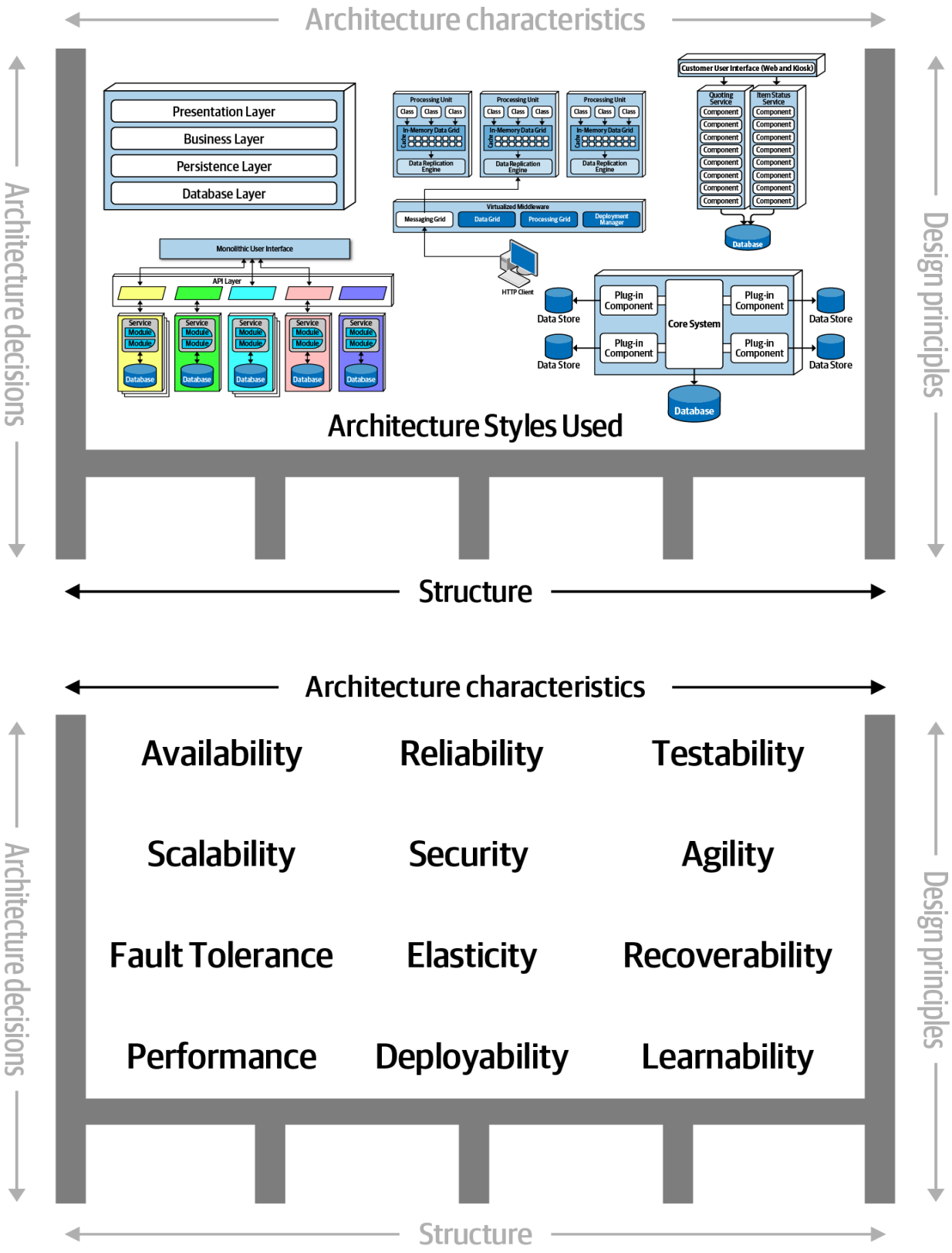
Defining Software Architecture

The industry as a whole has struggled to precisely define “software architecture.” Some architects refer to software architecture **as the blueprint of the system**, while others define it **as the roadmap for developing a system.**

The following figure illustrates a way to think about software architecture**. In this definition, software architecture consists of the structure of the system**, **combined with architecture characteristics** (“-ilities”) the system must support, **architecture decisions**, and finally **design principles**.



The structure of the system refers to the **type of architecture styles** the system is implemented in (such as microservices, layered, or microkernel). For example, suppose an architect is asked to describe an architecture, and that architect responds “it’s a microservices architecture.” Here, the architect **is only talking about the structure of the system**, but not the architecture of the system. **Knowledge of the architecture characteristics, architecture decisions, and design principles is also needed to fully understand the architecture of the system.**


Architecture characteristics
Architecture decisions
Design principles
Presentation Layer
Business Layer
Persistence Layer
Database Layer
Processing Unit
Class
In-Memory Data Grid
Data Replication Engine
Virtualized Middleware
Messaging Grid
Data Grid
Processing Grid
Deployment Manager
Customer User Interface (Web and Kiosk)
Quoting Service
Item Status Service
Component
Database
Monolithic User Interface
API Layer
Service
Module
HTTP Client
Data Store
Plug-in Component
Core System
Architecture Styles Used
Structure
Architecture characteristics
Availability
Reliability
Testability
Scalability
Security
Agility
Fault Tolerance
Elasticity
Recoverability
Performance
Deployability
Learnability
Architecture decisions
Design principles
Structure

Architecture characteristics are another dimension of defining software architecture. The architecture characteristics **define the success criteria of a system as they are required** in order for the system to **function properly.**

Architecture characteristics are so important that we've devoted several sections to understanding and defining them.

The next factor that defines software architecture is **architecture decisions**. **Architecture decisions define the rules and constraints for how a system should be constructed and direct the development teams on what is and what isn't allowed.** For example, an architect might make an architecture decision that only the business and services layers within a layered architecture can access the database, restricting the presentation layer from making direct database calls.

The last factor in the definition of architecture is **design principles.** A design principle ***differs from an architecture decision*** in that a **design principle is a guideline**. For example, a design principle states that the development teams should **leverage asynchronous messaging between services within a microservices architecture to increase performance**. An architecture decision (rule) **could never cover every condition and option** for communication between services, so **a design principle can be used to provide guidance for the preferred method** (in this case, asynchronous messaging).

When studying architecture, readers must keep in mind that, ***like much art, it can only be understood in context.*** Many of the decisions architects made **were based on realities of the environment they found themselves in**. For example, Imagine strolling into a 2002 data center and telling the head of operations "Hey, I have a great idea for a revolutionary style of architecture, where each service runs on its own isolated machinery, with its own dedicated database (microservices). So, that means I'll need 50 licenses for Windows, another 30 application server licenses, and at least 50 database server licenses." In 2002, trying to build an architecture like microservices would be inconceivably expensive.

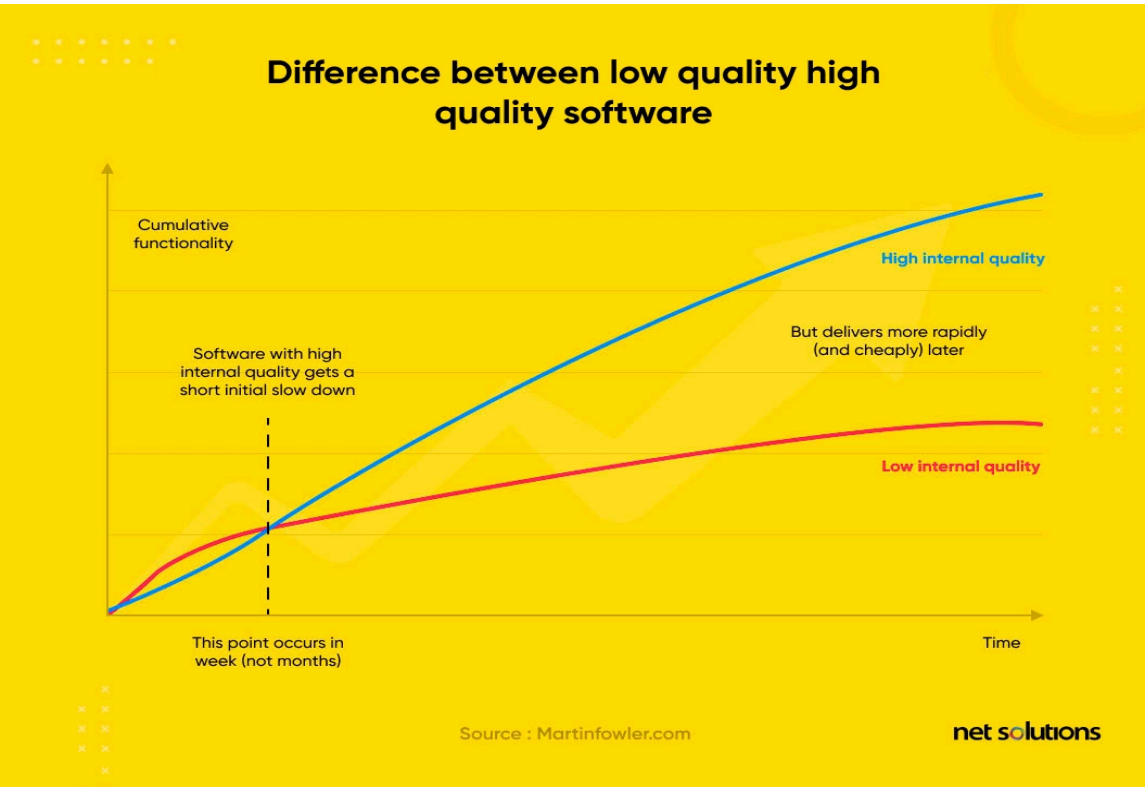
Readers should keep in mind that <u>all architectures are a product of their context</u>.

Why are Software Architecture Fundamentals Crucial ?

Software architects must make decisions within this constantly changing ecosystem. **Because everything changes, including foundations** upon which we make decisions, **architects should <u>reexamine some core axioms and norms</u>** that were informed earlier. For example, **earlier books about software architecture don't consider the impact of DevOps** because it didn't exist when these books were written.

Why Does Good Quality of Software Architecture Matters?

An organized software architecture helps to ensure the **longevity** of the software's **internal quality**.



Suppose you have two similar products that you launched within a three month gap. There are two scenarios:

- You launched Product A in Jan 2021. It supports a messy source code because the development team wanted to launch and monopolize the market as early as possible.

- You launched Product B in March 2021. It has a well-structured and organized software architecture. The development team worked on the design and architectural decisions early in the process, prioritizing quality over a faster launch.

Which Product will be more successful: A or B?

Product A might monopolize the market initially and convert better. However, product adoption will eventually subside because **the messy code will lead to technical debt pileups. These pile ups will, in turn, make it challenging to introduce new updates and bug fixes on the fly.**

On the other hand, Product B ***might have a market entry gap***, **but it will be easier to maintain a faster shipping cadence.** The customer needs will be looked after without breaking the shipping cadence, **thus making for a larger win.**

That is why the architecture in software engineering matters. It ensures you release high-quality software while maintaining a faster shipping cadence.

Good Software Architecture Goals

1. Defining a solution to meet requirements

Software strives to meet all functional, non-functional, technical, and operational requirements. Working closely with stakeholders, such as domain experts, business analysts, product owners, and end users, allows requirements to be identified and understood. A software architecture defines a solution that will meet those requirements. Poor architectures will lead to implementations that fail to meet the measurable goals of quality attributes, and they are typically difficult to maintain, deploy, and manage.

2. Balancing functional and non-functional requirements

Software architecture either enables quality attributes or inhibits them. Quality attributes are measurable and testable properties of a system. Some examples of quality attributes include maintainability, interoperability, security, and performance. They are non-functional requirements of a software system as opposed to its features, which are functional requirements. Quality attributes and how they satisfy the stakeholders of the system are critical, and software architecture plays a large role in ensuring that quality attributes are satisfied. The design of a software architecture can be made to focus on certain quality attributes at the cost of others (Trade-offs). Quality attributes may be in conflict with each other. A software architecture, when designed properly, sets out to achieve agreed-upon and validated requirements related to quality attributes.

3. Giving you the ability to predict software system qualities

When you look at a software architecture and its documentation, you can predict the software system's qualities. Making architecture decisions based on quality attributes makes it easier to fulfill those requirements. You want to start thinking about quality attributes as early as possible in the software development process as it is much more difficult (and costly) to make changes to fulfill them later. By thinking about them up front, and using modeling and analysis techniques, we can ensure that the software architecture can meet its non-functional requirements. If you are not able to predict if a software system will fulfill quality attributes until it is implemented and tested, then costly and time-consuming rework may be necessary. A software architecture allows you to predict a software system's qualities and avoid costly rework.

4. Easing communication among stakeholders

Software architecture and its documentation allow you to communicate the software architecture and explain it to others. It can form the basis for discussions related to aspects of the project, such as costs and duration. A software architecture is abstract enough that many stakeholders, with little or no guidance, should be able to reason about the software system. Although different stakeholders will have different concerns and priorities in terms of what they want to know about the architecture, providing a common language and architecture design artifacts allows them to understand the software system. It is particularly useful for large, complex systems that would otherwise be too difficult to fully understand. As requirements and other early decisions are made for the software system, a formal software architecture plays an important role and facilitates negotiations and discussions.

5. Managing change

Changes to a software system are inevitable. The catalyst for change can come from the market, new requirements, changes to business processes, technology advances, and bug fixes, among other things. Some view software architecture as inhibiting agility and would prefer to just let it emerge without up-front design. However, a good software architecture helps with both implementing and managing changes.

6. Providing a reusable model

An established architecture might be used again within an organization for other products in a product line, particularly if the products have similar requirements. When code is reused, resources, such as time and money, are saved. More importantly, the quality of software that takes advantage of reuse is increased because the code has already been tested and proven. The increase in quality alone translates to savings in resources. When a software architecture is reused, it is not just code that is reused. All of the early decisions that shaped the original architecture are leveraged as well. The thought and effort that went into the requirements necessary for the architecture, particularly non-functional requirements, may be applicable to other products. The effort that went into making those decisions does not necessarily have to be repeated. The experience gained from the original architectural design can be leveraged for other software systems. When a software architecture is reused, it is the architecture itself, and not just the software product, that becomes an asset to the organization.

7. Imposing implementation constraints

A software architecture introduces constraints on implementation and restricts design choices. This reduces the complexity of a software system and prevents developers from making incorrect decisions. If the implementation of an element conforms to the designed architecture, then it is abiding by the design decisions made by the

architecture. Software architecture, when done properly, enables developers to accomplish their objectives and prevents them from implementing things incorrectly.**Avoids Overengineering**: Focuses on robust but practical architecture to prevent unnecessary complexity.

8. Serves as training for team members

The system's architecture and its documentation serve as training for the developers on the team. By learning the various structures and elements of the system, and how they are supposed to interact, they learn the proper way in which the functionality is to be implemented. A software development team may experience change, such as having new team members join or existing ones leave. The introduction and orientation of new members to a team often takes time. A well-thought-out architecture can make it easier for developers to transition to the team. The maintenance phase of a software system can be one of the longest and costliest phases of a software project. Like new team members introduced during development, it is common for different developers to work on the system over time, including those introduced to maintain it. Having a solid architecture available to teach and bring aboard new developers can provide an important advantage.

Why is there no path for software architects?

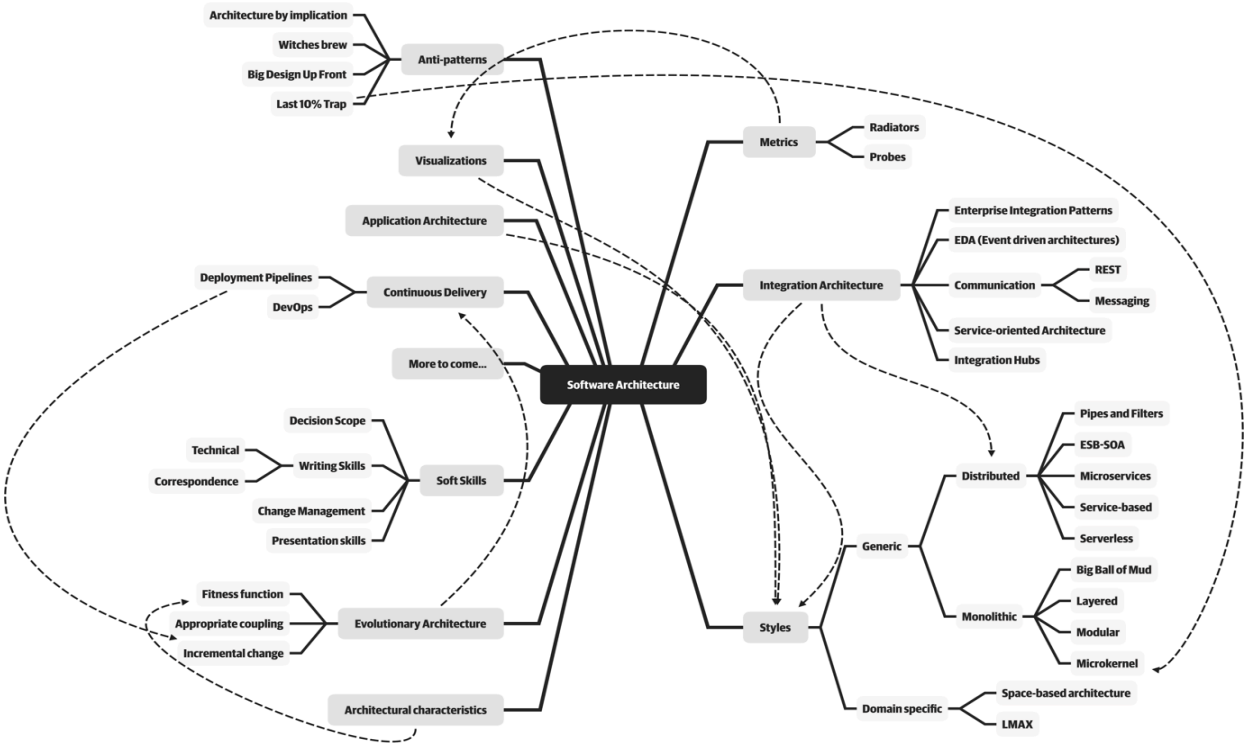
The job “software architect” appears near the top of numerous lists of best jobs across the world. Yet when readers look at the other jobs on those lists (like nurse practitioner or finance manager), there’s a clear career path for them.

So why do software architects have no clear path?

First, the industry doesn’t have a good definition of software architecture itself.

Second, the following figure illustrates in the mindmap, **the role of software architect embodies a massive amount and scope of responsibility that continues to expand.** A

decade ago, software architects **dealt only with the purely technical aspects** of architecture, like modularity, components, and patterns. Since then, **because of new architectural styles** (like microservices), **the role of software architect has expanded.**



Check also this architect mindmap at this link: https://xmind.app/embed/2R2h/

Third, **software architecture** is a **constantly moving target** because of the rapidly evolving software development ecosystem. **Any definition cast today will be hopelessly outdated in a few years. The Wikipedia definition of software architecture provides a reasonable overview, but many statements are outdated, such as "Software architecture is about making fundamental structural choices which are costly to change once implemented." as within microservices architecture, the idea of incremental built in—it is no longer expensive to make structural changes in microservices.**

Expectations of an Architect

Defining the role of a software architect presents as much difficulty as defining software architecture. **It can range from expert programmer up to defining the strategic technical direction** for the company. So it's recommended focusing on the expectations of an architect.

There are **seven core expectations** placed on a software architect, whatever the role, title, or job description:

- **Make architecture decisions**
- **Continually analyze the architecture**
- **Ensure compliance with decisions**
- **Diverse exposure and experience**
- **Have business domain knowledge**
- **Possess interpersonal skills**
- **Understand and navigate politics**

The first key to effectiveness and success in the software architect role depends on understanding and practicing each of these expectations.

Make Architecture Decisions

An architect is expected to define the architecture decisions and design principles used to guide technology decisions within the team, the department, or across the enterprise.

Guide is the key operative word in this first expectation.

An architect should guide rather than specify technology choices.

For example, an architect might make a decision to use React.js for frontend development. In this case, **the architect is making a technical decision rather than an architectural decision or design principle that will help the development team make choices**. An architect should **instead instruct** development teams to **use a reactive-based framework for frontend web development**, hence guiding the development team in making the choice between Angular, Elm, React.js, Vue, or any of the other reactive-based web frameworks.

Architects often struggle with finding the correct line of their architectural decisions.

The key to making effective architectural decisions is asking **whether** the architecture **decision is helping to guide teams in making the right technical choice or the architect makes the technical choice for them**. **Also an architect might need to make specific technology decisions in order to preserve a particular architectural characteristic** such as

scalability, performance, or availability**.** In this case it would **still be considered an architectural decision**.

Continually Analyze the Architecture

An architect is expected to continually analyze the architecture and current technology environment and then recommend solutions for improvement.

Other forgotten aspects of this expectation that **architects frequently forget are testing and release environments**. **Agility for code modification has obvious benefits, but if it takes teams weeks to test changes and months for releases, then architects cannot achieve agility in the overall architecture.**

Ensure Compliance with Decisions

An architect is expected to ensure compliance with architecture decisions and design principles.

Ensuring compliance means that the architect is **continually verifying that development teams are following the architecture decisions and design principles defined, documented, and communicated by the architect**. Consider the scenario where an architect makes a decision to restrict access to the database in a layered architecture to **only the business and services layers**. This means that the **presentation layer must go through all layers of the architecture to make even the simplest of database calls**. A user interface **developer** might **disagree with this decision** and **access the database directly for performance reasons**. **However, the architect made that architecture decision for a specific reason: to control change. By closing the layers, database changes can be made without impacting the presentation layer.** By not ensuring compliance with architecture decisions, **violations like this can occur**, the architecture will fail to meet the necessary characteristics.

In Chapter "Characteristics Governance and Fitness Functions" we talk more about measuring compliance using automated fitness functions and automated tools.

Diverse Exposure and Experience

An architect is expected to have exposure to multiple and diverse technologies, frameworks, platforms, and environments.

This expectation means **an architect must at least be familiar with a variety of technologies**. **Most environments** these days are **heterogeneous**, and at a minimum an architect should know **how to interface with multiple systems and services, irrespective of the language, platform, and technology those systems or services are written in.**

One of the best ways of mastering this expectation is for the architect to stretch their comfort zone. An effective software architect should be aggressive in seeking out opportunities to gain experience in multiple languages, platforms, and technologies. To focus on technical breadth rather than technical depth. Technical breadth includes the stuff you know about, but not at a detailed level, combined with the stuff you know a lot about.

Keep Current with Latest Trends

Also an architect is expected to keep current with the latest technology and industry trends.

Developers must keep up to date on the latest technologies they use on a daily basis to remain relevant (and to retain a job!).
An architect has an even more critical requirement to keep current on the latest technical and industry trends. The decisions an architect makes tend to be long-lasting and difficult to change.

Have Business Domain Knowledge

An architect is expected to have a **certain level of business domain expertise.**

Effective software architects understand not only technology **but also the business domain of a problem space. *Without business domain knowledge, it is difficult* to understand the business problem, goals, and requirements, *making it difficult* to design an effective architecture to meet the requirements of the business.** Imagine being an architect at a large financial institution and not understanding common financial terms such as an average directional index, aleatory contracts, rates rally, or even nonpriority debt. Without this knowledge, an architect cannot communicate with stakeholders and business users and will quickly lose credibility.

The most successful architects we know are those **who have broad, hands-on technical knowledge coupled with a strong knowledge of a particular domain. These software architects are able to effectively communicate with C-level executives and business users.** This in turn creates a strong level of confidence that the software architect knows what they are doing and is competent to create an effective and correct architecture.

Possess Interpersonal Skills

An architect is expected to possess **exceptional interpersonal skills**, including **teamwork, facilitation, and leadership.**

Having exceptional leadership and interpersonal skills is a difficult expectation for most developers and architects.
As technologists, developers and architects like to solve technical problems, **not people problems**. However, as Gerald Weinberg was famous for saying, “no matter what they tell you, **it’s always a people problem**.” An architect is also expected to **lead the development teams** through the implementation of the architecture.

The industry is flooded with software architects, all competing for a limited number of architecture positions. **Having strong leadership and interpersonal skills is a good way for an architect to differentiate themselves from other architects and stand out from the crowd.**

Understand and Navigate Politics

An architect is expected to understand the political climate of the enterprise and be able to navigate the politics.

To illustrate how important and necessary negotiation skills are, consider the scenario where an architect, responsible for a large CRM, faces issues with controlling database access, due to widespread database usage. To address this, the architect decides to create application silos, restricting each database to its owning application. This improves control over data, security, and changes. **However, this decision will likely be opposed by most of the company**, as other applications needing customer data will now have to access it through remote protocols like REST or SOAP.

The main point is that almost every decision an architect makes will be challenged by product owners, project managers, and business stakeholders due to increased costs or increased effort (time) involved. Architectural decisions will **also be challenged by developers who feel their approach is better.** In either case,

the architect must navigate the politics of the company and apply basic negotiation skills to get most decisions approved.

A software architect must combine business requirements and technical constraints to produce a high level architecture, and communicate this design between project stakeholders and developers. The traditional architect role, however, because of its distance to actual

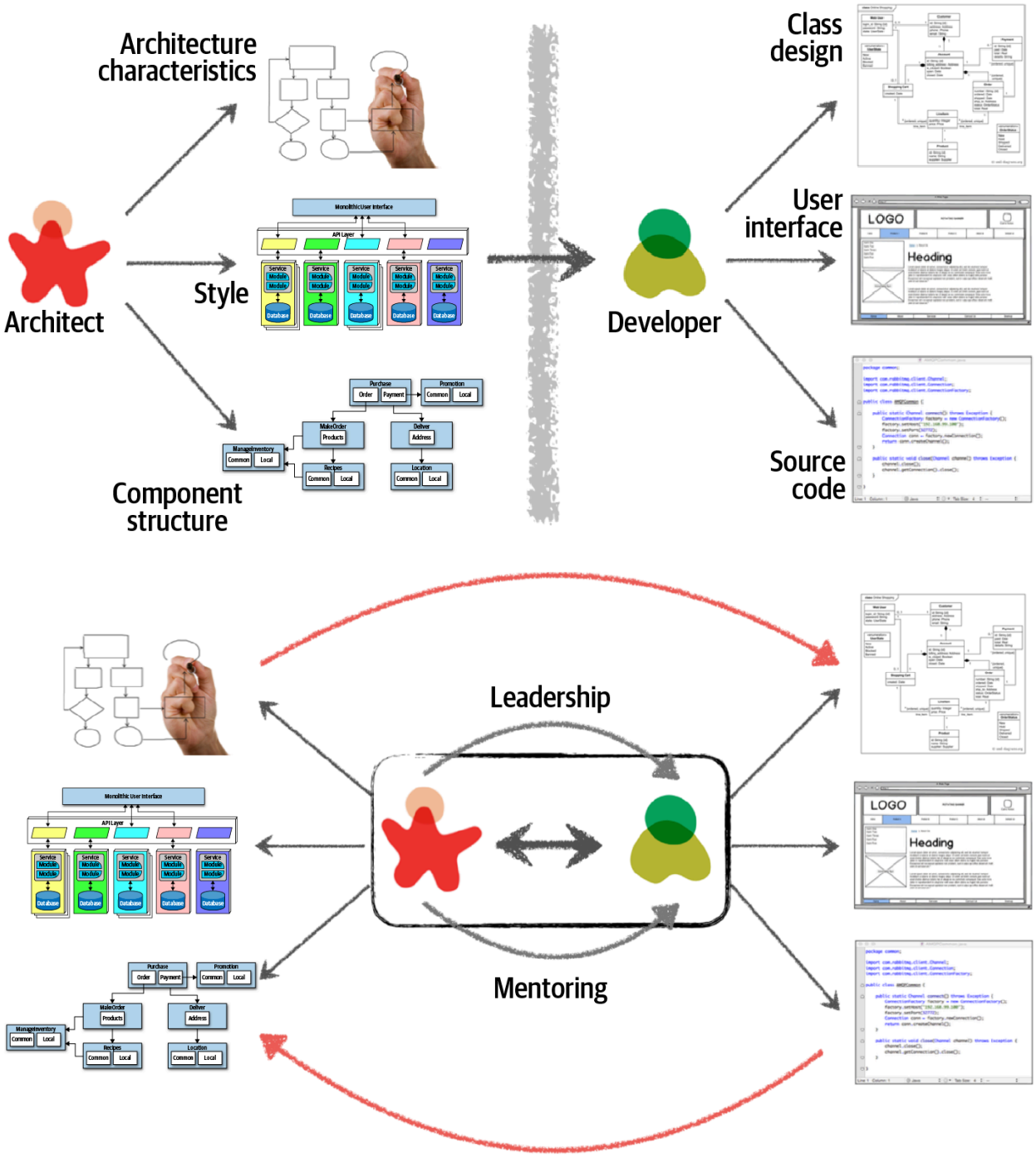
development, its technical skills and knowledge tends to become outdated (technology changes rapidily!) and will end up enclosing itself on an ivory tower and start pushing unrealistic or too rigid designs down the throat of developers. At some point they'll become bureaucrats, reading and preaching Gartner reports, attending to consulting-sponsored conferences and meeting other corporate architects. This may work for him if he is always on the same company and becomes a member of the "corporate confidence circle". But when the architect looks after a job position in other places he will often be questioned about his ability of getting things done.
A software developer is the other side of the coin. He's always in contact with the technical stuff and after gaining experience, he will be the company's reference on solving difficult problems. However career advancement for a developer will depend on (1) how unique and essential is his knowledge to the company, (2) his ability to talk with other stakeholders and (3) his ability to lead a team of less experienced developers. (1) depends on the developer commitment to some technology or paradigm and he must clearly stand-out to receive any recognition for it. (2) and (3) are natural skills any software architect must have.
With these assumptions, I believe the traditional role of a software architect is deemed to die or become obsolete. The model of a software developer with actual experience, bigger grasp in technology and ability to talk, negotiate and lead is the way to go.

Architectural Thinking and Planning

First you have to see things with an **architectural eye**, or an architectural point of view. **There are four main aspects of thinking like an architect**. **First, it's understanding the difference between architecture and design** and knowing how to collaborate with development teams to make architecture work. **Second, it's about having a wide breadth of technical knowledge while still maintaining a certain level of technical depth**, allowing the architect to see solutions and possibilities that others do not see. **Third, it's about understanding, analyzing, and reconciling trade-offs between various solutions and technologies. Finally, it's about understanding the importance of business drivers** and how they translate to architectural concerns.

Architecture vs Design

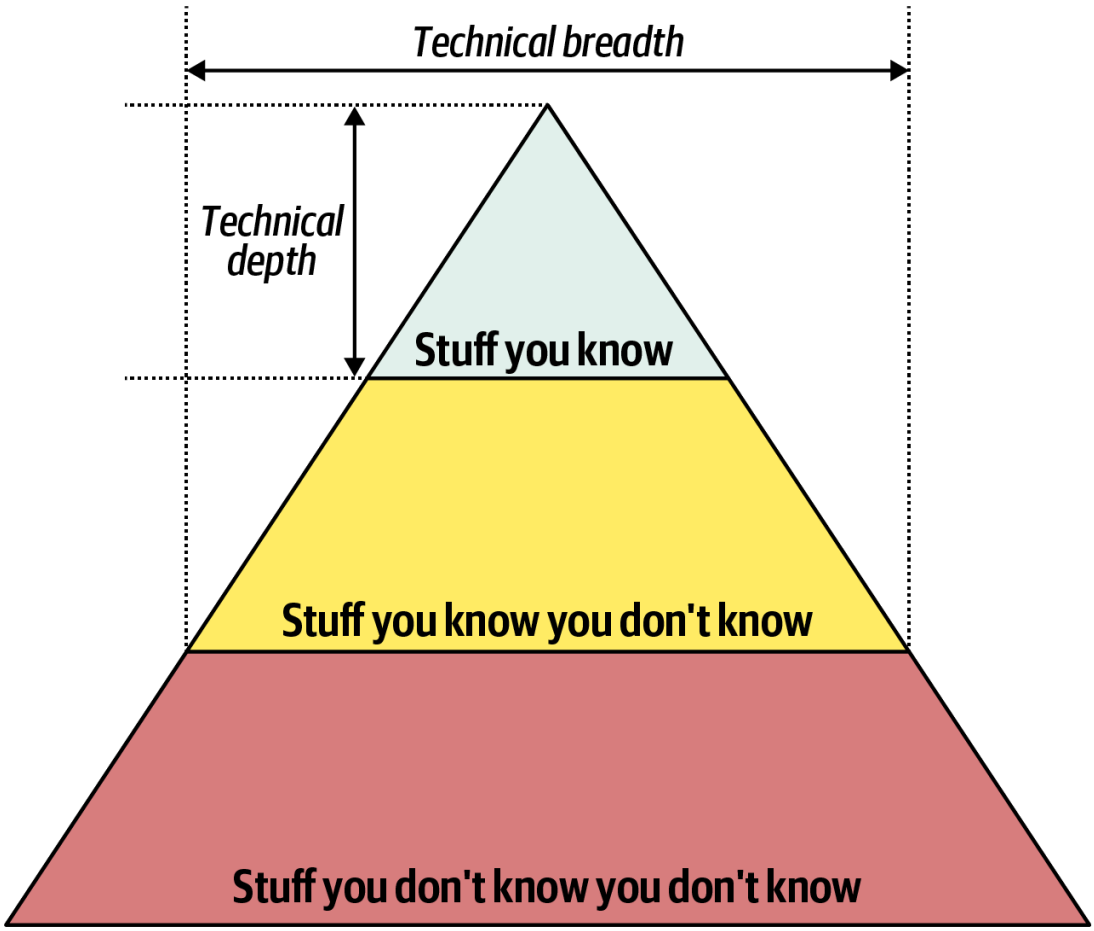
The difference between architecture and design is often a confusing one.



To make architecture work, both the **physical and virtual barriers** that exist between architects and developers **must be broken down**, thus forming a **strong bidirectional relationship between architects and development teams.** The architect and developer must be on the same virtual team to make this work, as depicted in Figure. Not only does this model facilitate strong bidirectional communication between architecture and development, but it also **allows the architect to provide mentoring and coaching to developers on the team.** Unlike the old-school waterfall approaches to static and rigid software architecture, the

architecture of today's systems changes and evolves every iteration or phase of a project. **A tight collaboration between the architect and the development team is essential for the success of any software project**.

Technical Breadth and Technical Depth



The scope of technological detail differs between developers and architects.

A developer must have a significant amount of technical depth to perform their job, **a software architect must have a significant amount of technical breadth to think like an architect and see things with an architecture point of view.**

As an architect, breadth is more important than depth. **Because architects must make decisions that match capabilities to technical constraints**, a broad understanding of a wide

variety of solutions is valuable. Thus, for an architect, the wise course of action is to sacrifice some hard-won expertise and use that time to broaden their portfolio.

For example, as an architect, it is more beneficial to know that five solutions exist for a particular problem than to have singular expertise in only one.

Developers transitioning to the architect role may have to change the way they view knowledge acquisition. Balancing their portfolio of knowledge regarding depth versus breadth is something every developer should consider throughout their career.

Analyzing Trade-Offs

Thinking like an architect is all about seeing trade-offs in every solution, technical or otherwise, and analyzing those trade-offs to determine what is the best solution.

Architecture is the stuff you can't Google.

Everything in architecture is a trade-off, which is why the famous answer to every architecture question in the universe is ***"it depends."*** While many people get increasingly annoyed at this answer, it is unfortunately true. You cannot Google the answer to whether REST or messaging would be better, or whether microservices is the right architecture style, because it does depend. **It depends on the deployment environment**, **business drivers, company culture, budgets, timeframes, developer skill set, and dozens of other factors**. Everyone's environment, situation, and problem is different, hence why architecture is so hard.

There are no right or wrong answers in architecture—only trade-offs.

Programmers know the benefits of everything and the trade-offs of nothing. Architects need to understand both.

The point here is that everything in software architecture has a trade-off: an advantage and disadvantage. Thinking like an architect is analyzing these trade-offs, then asking "which is more important: extensibility or security?" The decision between different solutions will always depend on the business drivers, environment, and a host of other factors.

Understanding Business Drivers

Thinking like an architect is understanding the business drivers that are required for the success of the system and **translating those requirements into architecture**

characteristics (such as scalability, performance, and availability). This is a challenging task that requires the architect to have some level of **business domain knowledge** and healthy, collaborative **relationships with key business stakeholders.**

Balancing Architecture and Hands-On Coding

One of the difficult tasks an architect faces is how to balance hands-on coding with software architecture. We firmly believe that every architect should code and be able to maintain a certain level of technical depth.

Bottleneck Trap

The first tip in striving for a balance between hands-on coding and being a software architect is avoiding the bottleneck trap. The bottleneck trap occurs when the architect has **taken ownership of code within the critical path of a project** (usually the underlying framework code) and **becomes a bottleneck to the team.**
If the architect becomes the only person capable of making changes or decisions regarding certain core parts of the codebase, it creates a single point of failure.
This happens because the architect is not a full-time developer and therefore must balance between playing the developer role (writing and testing source code) and the architect role (drawing diagrams, attending meetings, and well, attending more meetings).
It can lead to frustration among team members who are dependent on the architect's contributions to advance their work. Moreover, it may discourage ownership and initiative among other developers if they feel that significant parts of the project are "off-limits" or tightly controlled by the architect.

One way to avoid the bottleneck trap as an effective software architect is **to delegate the critical path and framework code to others on the development team** and then **focus on coding a piece of business functionality** (a service or a screen) one to three iterations down the road. **Three positive things** happen by doing this. **First, the architect is gaining hands-on experience writing production code *while no longer becoming a bottleneck on the team.* Second, the critical path and framework code is distributed to the development team** (where it belongs), **giving them ownership and a better understanding of the harder parts** of the system. **Third, and perhaps most important, the architect is writing the same business-related source code as the development team** and is therefore better **able to identify with the development team in terms of the pain they might be going through** with processes, procedures, and the development environment.

Remain Hands-on & Technical Depth

How can a software architect still remain hands-on and maintain some level of technical depth? There are some basic ways an architect can still remain hands-on at work ***without***

having to "practice coding from home" (although it's recommended practicing coding at home as well).

The first way is to **do frequent proof-of-concepts or POCs.** For example, if an architect is **stuck trying to make a decision between two caching solutions**, one effective way **to help make this decision is to develop a working example in each caching product and compare the results.** This allows the architect to see **first-hand the implementation details and the amount of effort required to develop the full solution**. It also allows the architect to **better compare architectural characteristics** such as scalability, performance, or overall fault tolerance of the different caching solutions.

Another way an architect can remain hands-on is to tackle some of the technical debt stories or architecture stories**, freeing the development team up to work on the critical functional user stories. These stories are usually low priority**, so if the architect does not have the chance to complete a technical debt or architecture story within a given iteration, generally does not impact the success of the iteration.

A final technique to remain hands-on as an architect is to **do frequent code reviews**. While the architect is not actually writing code, **at least they are involved in the source code** and to seek out mentoring and coaching opportunities on the team.

Software Architecture Principles (Set of Best Practices)

Software Architecture Principles aim to guide the development of systems that are **robust, maintainable, scalable, and adaptable.** They focus on **ensuring system quality**, **facilitating ease of maintenance**, **enhancing scalability**, promoting component **reusability**, supporting system flexibility and **adaptability**, guiding **effective system decomposition**, encouraging **consistency and standardization** across the development process, **improving stakeholder communication**, **optimizing resource utilization**, and **reducing risks** associated with software development.

This guide delves into the software architecture principles **spanning from High-Level Design (HLD) to Low-Level Design (LLD)**, highlighting some principles, such as the **Separation of Concerns** (SoC).

Separation of Concerns (SoC)

The separation of concerns (SoC) is one of the most fundamental principles in software development.
It is so crucial that 2 out of 5 SOLID principles (Single Responsibility and Interface Segregation) are direct derivations from this concept.
The principle is simple: don't write your program as one solid block, instead, break up the code into chunks that are finalized tiny pieces of the system each able to complete a simple distinct job.

SoC is particularly important in software architecture for several reasons:

- Manageability: It reduces the complexity of managing large software projects by breaking down the system into manageable, understandable parts.
- Maintainability: Changes, updates, or fixes can be made in one part of the system without impacting others, reducing the risk of unintended side effects.
- Scalability: Independent components can be scaled in isolation, allowing for more efficient resource use and performance optimization.
- Testability: Isolating concerns enhances the ability to test features in isolation, improving test coverage and making debugging easier.
- Reusability: Modular components can be reused across different parts of the system or in different projects, reducing development time and cost.

SoC for programming functions (Low Level Design)

If we take the lowest level (the actual programming code), SoC instructs us to avoid writing long complex functions. When the function starts to bloat up in size, this is the red flag that the method is possibly taking care of too many things at once.
In such a case SoC pushes us to refactor it, turning into a more laconic and descriptive revision. During this process, parts of the original algorithm get exported and encapsulated in separate smaller functions with a private access level. We gain the code clarity, and chunks of the algorithm eventually become reusable by other parts, even if we initially didn't expect this to happen.

SoC Modularity (High Level Design)

*95% of the words [about software architecture] are spent extolling the benefits of "modularity" and that **little is said about how to achieve it**.*

Modularity is an organizing principle that serves as a structural approach to implementing the Separation of Concerns (SoC) principle.
We use modularity to **describe a logical grouping of related code**, which could be a group of classes in an **object-oriented language** or functions in a structured or functional language.
Most languages provide mechanisms for modularity (namespace in .NET, package in Java, and so on).
If an architect designs a system without paying attention to how the pieces wire together, they end up creating a system that presents myriad difficulties.

Architects must be aware of how developers package things because it has important implications in architecture. For example, if several packages are tightly coupled together, reusing one of them for related work becomes more difficult.
Object-oriented languages became popular because they offered new ways to encapsulate and reuse code.

For discussions about architecture, we use modularity as a general term to denote a related grouping of code: classes, functions, or any other grouping. This doesn't imply a physical separation, merely a **logical one**; the difference is sometimes important.

How to measure Modularity?

Cohesion

It is a measure of how closely related the parts are to one another; and closely related means in terms of **common goals, shared data and functional dependency.**
A cohesive module is one where all the parts should be packaged together.
Attempting to divide a cohesive module would only result in increased coupling and decreased readability as you have to set up calls between modules to achieve useful results and

That's why **high cohesion** is **good**.

Cohesion Measures: Sorted from High (Best) to Low (Worst) Cohesion

- **Functional cohesion:**
  Every part of the module is related to the other, and the module contains everything essential to function.
- **Sequential cohesion:**
  Two modules interact, where one outputs data that becomes the input for the other.
- **Communicational cohesion:**
  Two modules form a communication chain, where each operates on information and/or contributes to some output. For example, add a record to the database and generate an email based on that information.
- **Procedural cohesion:**
  Two modules must execute code in a particular order.
- **Temporal cohesion:**
  Modules are related based on timing dependencies. For example, many systems have a list of seemingly unrelated things that must be initialized at system startup; these different tasks are temporally cohesive.
- **Logical cohesion:**
  The data within modules is related logically but not functionally. For example, consider a module that converts information from text, serialized objects, or streams. Operations are related, but the functions are quite different. A common example of this type of cohesion exists in virtually every Java project in the form of the StringUtils package: a group of static methods that operate on String but are otherwise unrelated.
- **Coincidental cohesion:**
  Elements in a module are not related other than being in the same source file; this represents the most negative form of cohesion.

For example, consider this module definition:

- *Customer Maintenance: add customer, update customer, get customer, notify customer,* ***get customer orders, cancel customer orders.***

Should the **last two entries reside in this module or should the developer create two separate modules**, such as:

- *Customer Maintenance: add customer, update customer, get customer, notify customer*
- *Order Maintenance: get customer orders, cancel customer orders*

Which is the correct structure? As always, it depends:

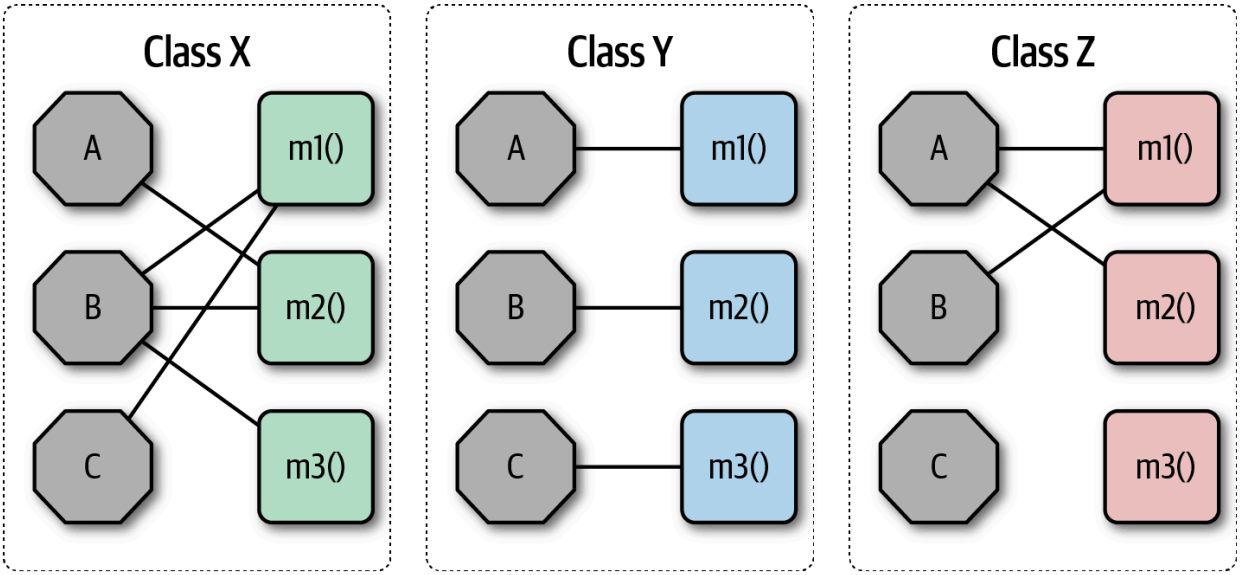
Are those the **only two operations for Order Maintenance?** If so, it may make sense to collapse those operations back into Customer Maintenance.

Is Customer Maintenance expected to grow much larger, encouraging developers to look for opportunities to extract behavior?

Does Order Maintenance require so much knowledge of Customer information that separating the two modules would require a high degree of coupling to make it functional? This relates back to the Larry Constantine quote.

A well-known set of metrics named the Chidamber and Kemerer Object-oriented metrics suite was developed by the eponymous authors to measure particular aspects of object-oriented software systems.
The suite includes many common code metrics, such as **cyclomatic complexity** and several important **coupling metrics**.

Consider these three classes:



In the figure above, fields appear as single letters and methods appear as blocks. **In Class X, Many methods are sharing shared fields**, indicating **good structural cohesion**. **Class Y**, however, **lacks cohesion**; each of the field/method pairs in Class Y could appear in its own class without affecting behavior. **Class Z** shows **mixed cohesion**, where developers could refactor the last field/method combination into its own class.

Coupling

It describes the degree of direct knowledge or **dependency one module has on another**, indicating how closely connected different modules are within a system.
Focus: **The interactions between different modules.**
Goal: **Loose coupling between modules is desirable**. It means that changes in one module are less likely to require changes in another. **Modules** can be understood, **developed**, **and**

updated more **independently**, **enhancing the system's modularity.**

Afferent and Efferent Coupling

It is including the metrics afferent and efferent coupling. ***Afferent coupling*** measures the number of **incoming connections to a Single code artifact** (component, class, function, and so on). ***Efferent coupling*** measures the **outgoing connections from this Single code artifact** to other **code artifacts.**

Decoupling

It is intrinsically linked to the concept of coupling, serving as its countermeasure. **By minimizing coupling**, decoupling aims to achieve several key architectural benefits:

- **Enhanced Modularity**: Decoupling promotes the development of modules or components that can be developed, tested, and deployed independently. This modularity is crucial for managing large-scale systems and facilitates parallel development across different teams.
- Improved Flexibility: With decoupled systems, it becomes easier to replace or update components without affecting the rest of the system.
- Increased Maintainability: **Loosely coupled systems are easier to maintain** since the impact of changes is localized. This isolation **reduces the risk of unintentional side effects on other parts of the system.**
- Scalability: Decoupling allows components to **scale independently**, **addressing performance bottlenecks more efficiently without requiring a scale-up of the entire system.**
- Fault Isolation: In decoupled architectures, failures in one component are less likely to propagate through the system, enhancing overall reliability and availability.

Connascence

Connascence is a more detailed and nuanced concept that goes beyond the idea of *coupling* to describe the types and strength of dependencies between different parts of the codebase.

Two components are connascent if a change in one would require the other to be modified in order to maintain the overall correctness of the system.

You can read and learn more about it in the book of Fundamentals of Software Architecture by Richard Mark & Neal Ford.

Decoupling Strategies

To achieve decoupling, several strategies can be employed, such as:

- **Interface-based Design**: Defining clear interfaces for components, which specify what a component does without dictating how it does it, allowing for flexibility in implementation, such as Facade and Proxy design patterns.
- **Dependency Injection**: Passing dependencies to objects instead of hardcoding them, making it easier to replace or modify those dependencies without altering the dependent objects.
- **Publish-Subscribe Patterns and Event-Driven Architectures**: Allowing components to communicate indirectly through events or messages, reducing the need for direct interactions and dependencies such as Message Queue, Event Bus and Observer Pattern.
- **Microservices Architecture**: Structuring applications as collections of loosely coupled services, each implementing a specific business function and communicating through well-defined APIs.

Abstractness, Instability, and Distance from the Main Sequence

Abstractness is the ratio of abstract artifacts (abstract classes, interfaces, and so on) **to concrete artifacts** (implementation). It represents a measure of **abstractness versus implementation**. For example, consider a code base with no abstractions, just a huge, single function of code. The flip side is a code base with too many abstractions, making it difficult for developers to understand how things wire together (for example, it takes developers a while to figure out what to do with an AbstractSingletonProxyFactoryBean).

Equation. **Abstractness**

$$A = \frac{\sum m^a}{\sum m^c + \sum m^a}$$

In the equation, *ma* represents **abstract elements** (interfaces or abstract classes) with the module, and *mc* represents **concrete elements** (nonabstract classes). This metric looks for the same criteria. The easiest way to visualize this metric: **consider an application with 5,000 lines of code, all in one main() method**. **The abstractness numerator is 1**, **while the denominator is 5,000**, yielding an **abstractness of almost 0**. Thus, this metric **measures the ratio of abstractions in your code**.

Another derived metric, instability, is defined as **the ratio of efferent coupling** to the sum of both efferent and afferent coupling, shown in Equation.

Equation. **Instability**

$$I = \frac{C^e}{C^e + C^a}$$

In the equation, *ce* represents **efferent (or outgoing) coupling**, and *ca* represents **afferent (or incoming) coupling.**

The instability metric determines the volatility of a code base. A code base that exhibits ***high degrees of instability breaks more easily*** when changed **because of high coupling**. For example, **if a class calls to many other classes** to delegate work, the calling class shows high susceptibility to breakage if one or more of the called methods change.

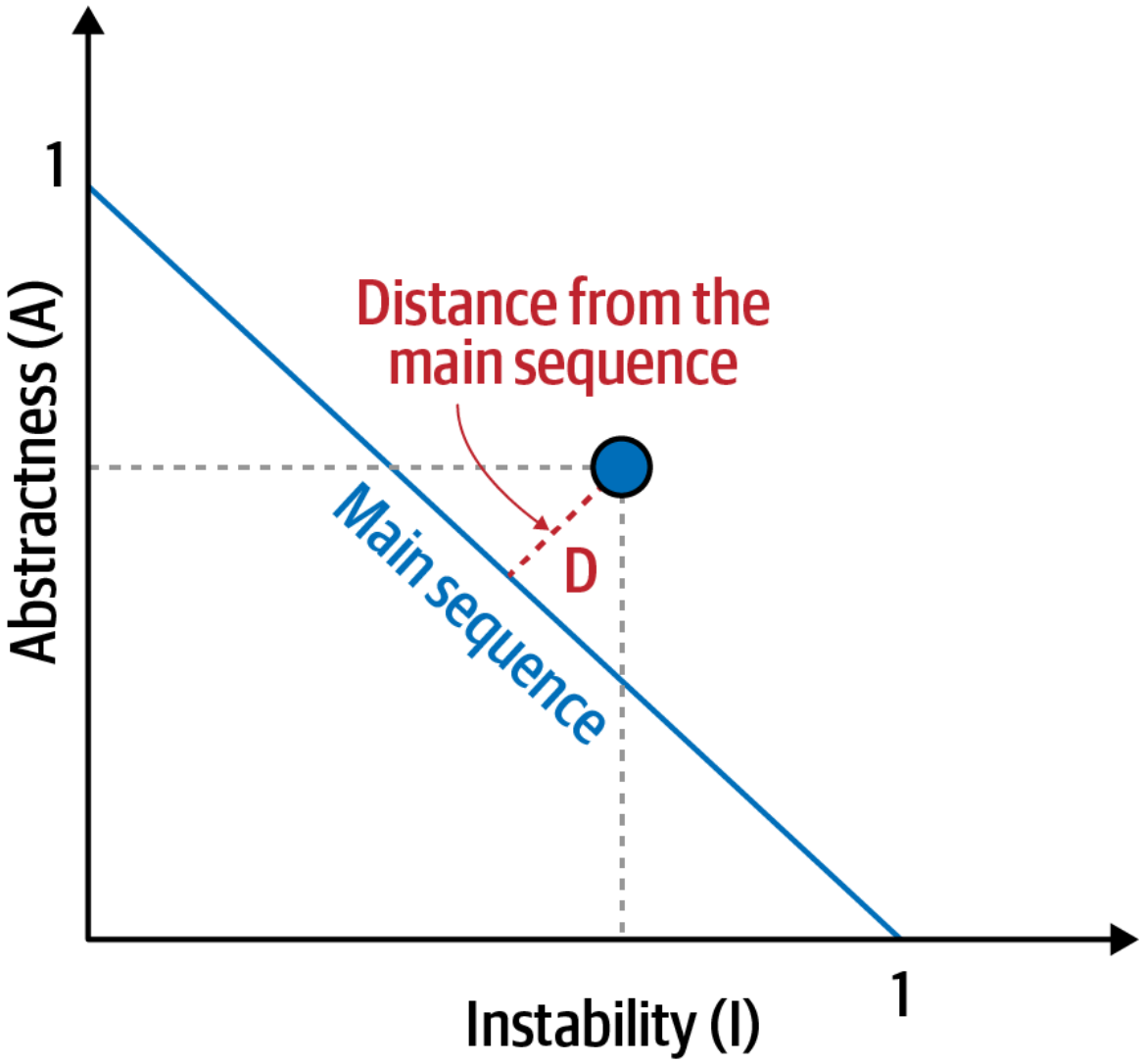
Distance From the main sequence

$$D = |A + I - 1|$$

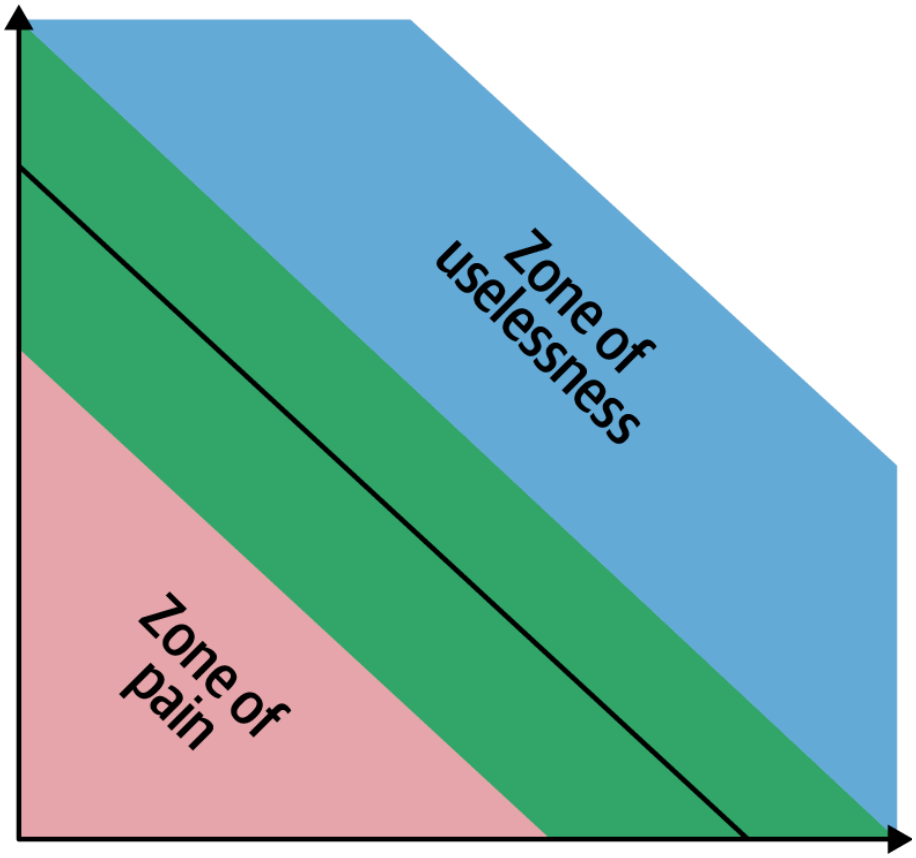
In the equation, A = abstractness and I = instability.

Note that both abstractness and instability are fractions whose results will always fall between 0 and 1.

The main sequence defines the ideal relationship between abstractness and instability.



In the figure, **developers graph the candidate class**, **then measure the distance from the idealized line. The closer to the line, the better balanced the class**. **Classes that fall too far into the upper-righthand corner enter into what architects call the zone of uselessness**: code that is too abstract becomes difficult to use. **Conversely, code that falls into the lower-lefthand corner enter the zone of pain**: code with too much implementation becomes brittle and hard to maintain,



<u>Tools exist</u> in many platforms to provide these measures, (such as NDepend Library in .Net) , which assist architects when analyzing code bases because of unfamiliarity, migration, or technical debt assessment.

Summary SoC vs Modularity vs SRP

Level of Abstraction:

- Separation of Concerns (SoC) represents the highest level of abstraction, guiding the overarching strategy for dividing a software system into distinct sections that separate the different concerns or functionalities.
- Modularity serves as the architectural implementation of SoC, detailing the division of a system into modules or components that encapsulate specific functionalities or concerns, thereby defining the system's structure.
- Single Responsibility Principle (SRP), the most granular among these principles, focuses on ensuring that each class or component within a module is tasked

with a single responsibility, thereby guiding the design and implementation within the modules.

Design Focus:

- SoC emphasizes the broad separation of functionalities and concerns across the entire software system, aiming to simplify overall system design and maintenance.
- Modularity concentrates on the structural aspects of software design, outlining how functionalities encapsulated by SoC are organized into discrete, interacting modules. This approach aids in achieving a clear and manageable system structure.
- SRP is concerned with the internal coherence of modules, specifying that each class or component should have only one reason to change. This principle directly influences the design and development of the classes and components themselves, ensuring clarity and simplicity within each module.

Application Scope:

- SoC applies to the entire spectrum of software design and architecture, offering a high-level principle for organizing different aspects of a software system.
- Modularity focuses specifically on the architectural configuration of the system, advocating for a design that breaks down the system into a set of well-defined, loosely coupled modules.
- SRP targets the internal design of individual components or classes within these modules, ensuring that each has a single, well-defined purpose. By adhering to SRP, developers can create more robust, understandable, and maintainable components.

In essence, while SoC, Modularity, and SRP share the common goal of managing complexity and improving system maintainability, they operate at different levels of abstraction and scopes within the software design process, from the high-level organization of concerns to the detailed responsibilities of individual components.

Bounded Context (Domain Driven Design)

Bounded Context is a core concept in Domain-Driven Design (DDD) that plays a pivotal role in managing complexity within large software systems, **particularly those with intricate business logic**. This principle is instrumental in organizing the system's domain model and its interaction with the external world, ensuring clarity and consistency in how business concepts are applied and implemented in software.

Understanding Bounded Context

A Bounded Context is primarily a linguistic delimitation, that is to say that terms and sentences can mean different things, according to the context in which they are employed.
This linguistic delimitation refers to ubiquitous language, which is another essential element in DDD.

A Bounded Context in Domain-Driven Design (DDD) **sets boundaries** within which specific terms maintain consistent meanings, addressing the challenge of varying interpretations across a large system. This concept emphasizes the importance of an **"ubiquitous language" — a shared vocabulary that ensures clear communication** among all project stakeholders, including developers, domain experts, and users.

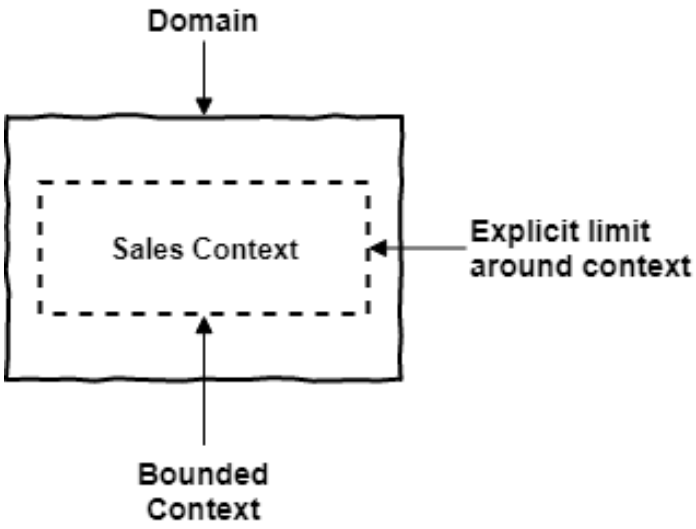
In software architecture, this principle is vital for reducing ambiguities and fostering a **clear understanding and separation** across different parts of the system, **making it easier to manage complexity** and **integrate** various components **effectively**.

As a software architecture principle, **Bounded Context is crucial for managing complexity in large systems**. It allows architects to:

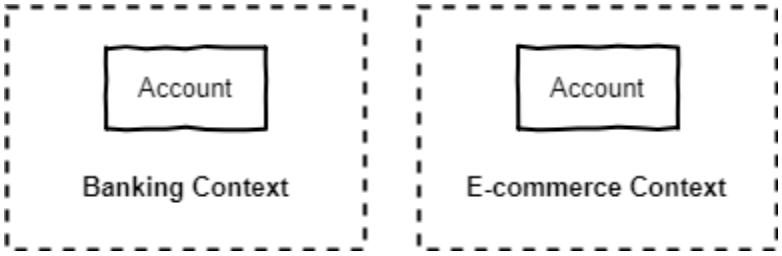
- **Divide the system into manageable parts**: Each bounded context focuses on a specific aspect of the business, making it easier to manage and evolve.
- **Ensure consistency within contexts**: By using a ubiquitous language, each bounded context maintains internal consistency, with terms having clear, agreed-upon meanings.
- **Facilitate integration between contexts**: Defined boundaries and languages make it easier to design interactions between different parts of the system, as each interaction can be clearly understood in terms of the involved contexts' languages.

– In a forthcoming section “Microservice Decomposition”, we will delve into a detailed case study focusing on the transition from a monolithic architecture to a microservices-based approach, aimed at improving both maintainability and scalability. This case study will highlight the strategic use of Bounded Context as a methodological framework for facilitating this migration, offering insights into its practical application and benefits in modern software development practices.

It is also important to understand that **Bounded Context is where the Model is implemented**, that is, a Bounded Context is the solution implementation in a technical way.



Practical example



Look at the image above.

You may have noticed that there are two Bounded Contexts and within them there is an ‘Account‘ entity.
It is easy to identify that ‘Account‘ has different meanings in these two contexts.
In the banking context ‘Account‘ refers to subjects such as money, transactions, payments, credit, and may belong to a person or company.
In the context of e-commerce, ‘Account‘ refers to subjects such as login, account creation, password change and so on.
With this information we can identify in which context the entity ‘Account‘ belongs even if it had not been told which context it belongs here.
However, reality is often different, and the same entity will be in distinct Bounded Contexts within the same company, and in subjects that look the same. Be very careful with that!

Bounded Contexts and Subdomains

Differences

It is only natural when we start studying DDD that we find content that makes **it difficult to differentiate between Bounded Contexts and Subdomains**, since they both represent “**segregation**”.
Vaughn Vernon in his book “Implementing Domain-Driven Design” states that **Subdomains live in the space of the problem** and the **Bounded Contexts in the solution space**.
With this it is clear to differentiate that, **Subdomains are logical “separations”** of the domain and **Bounded Contexts are technical solutions**.

Relationship

It is desirable that code belonging to **a Bounded Context implements a single Subdomain**.
So we **segregate** Domain Models by real business intent.
However, **this relationship is not always possible**, the most frequent reason is when we are working on legacy systems built without the DDD approach, where a single Bounded Context implements more than one Subdomain, this can generate the Big Ball of Mud, that has absence of any discernible architecture structure.

Teams and Source Code Repositories

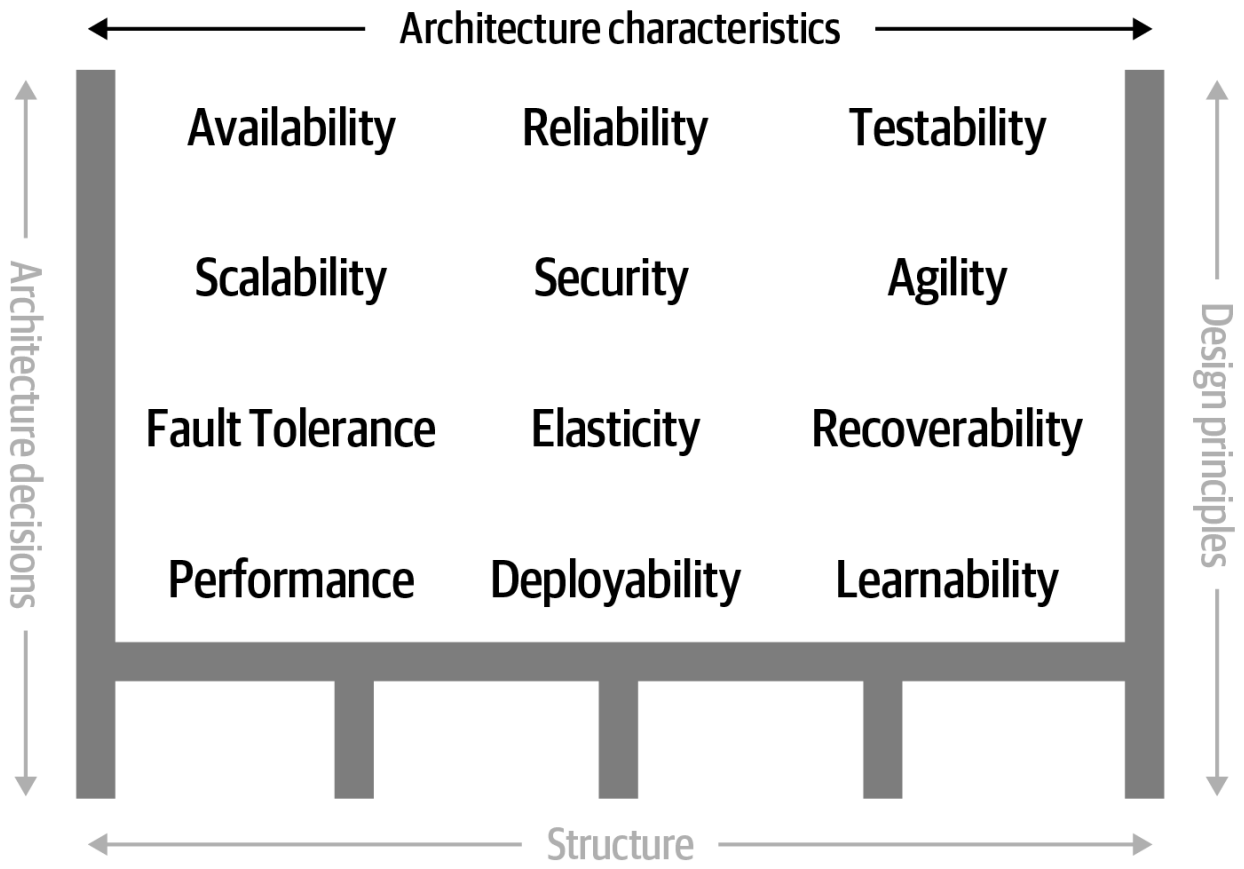
I really like the definitions that “Vaughn Vernon – Domain-Driven Design Distilled“, states in his book in summary form:

- There must be a team assigned to work on a Bounded Context.
- There should be a separate source code repository for each Bounded Context.
- It is possible that one team can work in several Bounded Context, but several teams should not work in a single Bounded Context.

Architecture characteristics

Architects may collaborate on defining the domain or business requirements, **but as a must, one key responsibility entails defining, discovering, and analyzing architectural characteristics** as it isn't directly related to the domain functionality.

What distinguishes software architecture from coding and design? Many things, including the role that **architects** have in **defining architectural characteristics**, the important aspects of the system independent of the problem domain.



An architecture characteristic meets three criteria:

Specifies a nondomain design consideration

The requirements specify what the application should do; **architecture characteristics specify operational and design criteria for success**, Concerning **how to implement the requirements** and **why certain choices were made**.
For example, a common important architecture characteristic specifies a **certain level** of **performance** for the application, which often doesn't appear in a requirements document. Even more pertinent: no requirements document states "prevent technical debt," but it is a common design consideration for architects and developers.

Influences some structural aspect of the design

The primary reason to describe the characteristics is : does this architecture characteristic require special structural consideration and modifications to succeed? If Yes then it is a crucial one to address**.**
For example, security is a concern in virtually every project. However, it rises to the level of architectural characteristic when the architect needs to design something special for the requirements. In such cases, the security characteristic may require special structural modifications to ensure the success criteria of the design.
Consider these two cases surrounding payment:

- Third-party payment processor
  If an integration point handles payment details, then the architecture shouldn't require special structural considerations. The design should incorporate standard security hygiene, such as encryption and hashing, but doesn't require special structure.
- In-application payment processing
  If the application under design must handle payment processing, the architect may design a specific module, component, or service for that purpose to isolate the critical security concerns structurally. Now, the architecture characteristic has an impact on both architecture and design.

Is critical or important to application success

Support for each architecture characteristic adds complexity to the design and perhaps structure too. Moreover, **a bigger problem** lies with the fact that some characteristics can **conflict with others**. For example, if an architect wants to **improve security, it will almost certainly negatively impact performance**: the application must do more on-the-fly encryption, indirection for secrets hiding, and other activities that potentially degrade performance.
Thus, **a critical job for architects lies in making trade-offs between several competing concerns**, **choosing the fewest critical** architecture **characteristics** rather than the most possible.

Architecture Characteristics Categorized and Partially Listed

Architects commonly separate architecture characteristics into broad categories. The following sections describe a few, along with some examples.

Operational Architecture Characteristics

Operational architecture characteristics heavily overlap with operations and DevOps concerns, forming the intersection of those concerns in many software projects.

Table 4-1. Common operational architecture characteristics

Term	Definition
Availability	How long the system will need to be available (if 24/7, steps need to be in place to allow the system to be up and running quickly in case of any failure).
Continuity	Disaster recovery capability.
Performance	Includes stress testing, peak analysis, analysis of the frequency of functions used, capacity required, and response times. Performance acceptance sometimes requires an exercise of its own, taking months to complete.
Recoverability	Business continuity requirements (e.g., in case of a disaster, how quickly is the system required to be on-line again?). This will affect the backup strategy and requirements for duplicated hardware.
Reliability/safety	Assess if the system needs to be fail-safe, or if it is mission critical in a way that affects lives. If it fails, will it cost the company large sums of money?
Robustness	Ability to handle error and boundary conditions while running if the internet connection goes down or if there’s a power outage or hardware failure or incorrect inputs or unexpected user behavior or any external system changes.

Term	Definition
Scalability	Ability for the system to perform and operate as the number of users or requests increases. It has two types : Vertical or Horizontal Scaling.
Elasticity	Ability for the system to automatically or dynamically scale resources up or down as needed, in response to changing demand. It is a more dynamic concept than scalability.

Structural Architecture Characteristics

Architects must concern themselves with code structure. In many cases, the architect has sole or shared responsibility for code quality concerns, such as good modularity, controlled coupling between components, readable code, and other quality assessments.

Table 4-2. Structural architecture characteristics

Term	Definition
Configurability	Ability for the end users to easily change aspects of the software's configuration (through usable interfaces).
Extensibility	How important it is to plug new pieces of functionality in.
Installability	Ease of system installation on all necessary platforms.
Leverageability/reuse	Ability to leverage common components across multiple products.

Term	Definition
Localization	Support for multiple languages on entry/query screens in data fields; on reports, multibyte character requirements and units of measure or currencies.
Maintainability	How easy is it to apply changes and enhance the system?
Portability	Does the system need to run on more than one platform? (For example, does the frontend need to run against Oracle as well as SAP DB?
Upgradeability	Ability to easily/quickly upgrade from a previous version of this application/solution to a newer version on servers and clients.

Cross-Cutting Architecture Characteristics

Many other architecture characteristics fall outside categorization yet form important design constraints and considerations.

Table 4-3. Cross-cutting architecture characteristics

Term	Definition
Accessibility	Access to all your users, including those with disabilities like colorblindness or hearing loss.
Archivability	Will the data need to be archived or deleted after a period of time? (For example, customer accounts are to be deleted after three months or marked as obsolete and archived to a secondary database for future access.)
Authentication	Security requirements to ensure users are who they say they are.

Term	Definition
Authorization	Security requirements to ensure users can access only certain functions within the application (by use case, subsystem, webpage, business rule, field level, etc.).
Legal	What legislative constraints is the system operating in (data protection, Sarbanes Oxley, GDPR, etc.)? What reservation rights does the company require? Any regulations regarding the way the application is to be built or deployed?
Privacy	Ability to hide transactions from internal company employees (encrypted transactions so even DBAs and network architects cannot see them).
Security	Does the data need to be encrypted in the database? Encrypted for network communication between internal systems? What type of authentication needs to be in place for remote user access?
Supportability	What level of technical support is needed by the application? What level of logging and other facilities are required to debug errors in the system?
Usability/achievability	Level of training required for users to achieve their goals with the application/solution. Usability requirements need to be treated as seriously as any other architectural issue.

Note:

Many of the definitions overlap and no complete list of standards exists. For example, consider availability and reliability, which seem to overlap in almost all cases.

A consistent frustration amongst architects is the lack of clear definitions of so many critical things, including the activity of software architecture itself!
However, we do follow and **recommend** the advice from **domain-driven design** to establish and **use a *ubiquitous language* amongst fellow employees to help ensure fewer term-based misunderstandings.**

Identifying Architectural Characteristics

An architect uncovers architecture characteristics in at least **three ways** by extracting from implicit domain knowledge as discussed in the domain concerns, requirements, and. We previously discussed implicit characteristics and we cover the other two here.

Extracting Architecture Characteristics from Domain Concerns

An architect must be able to **translate domain concerns** to identify the right architectural characteristics.
Understanding the key domain goals and domain situation allows an architect to translate those domain concerns to "-ilities," which then forms **the basis for correct and justifiable architecture decisions.**
One similar tip to what we discussed before when **collaborating with domain stakeholders to define the driving architecture characteristics** is to work hard to keep the final list as short as possible.
A better approach is to have the domain stakeholders select the top three most important characteristics from the final list (in any order).
One problem is that architects and domain stakeholders speak different languages.
Architects talk about scalability, interoperability, fault tolerance, etc. Domain stakeholders talk about mergers and acquisitions, user satisfaction, time to market, and competitive advantage. Architects have no idea how to create an architecture to support user satisfaction, and domain stakeholders don't understand why there is so much focus and talk about availability, interoperability, etc.
Fortunately, there is usually a **translation from domain concerns to architecture characteristics**.

Table 5-1. Translation of domain concerns to architecture characteristics

Domain concern	Architecture characteristics

Mergers and acquisitions	Interoperability, scalability, adaptability, extensibility
Time to market	Agility, testability, deployability
User satisfaction	Performance, availability, fault tolerance, testability, deployability, agility, security
Competitive advantage	Agility, testability, deployability, scalability, availability, fault tolerance
Time and budget	Simplicity, feasibility

There is a trap many architects fall into when translating domain concerns. ***Focusing on only one of the ingredients*** is like forgetting to put the flour in the cake batter.

For example, a domain stakeholder might say something like “Due to regulatory requirements, it is absolutely imperative that we complete end-of-day fund pricing on time.” ***An ineffective architect might just focus on performance*** as it seems to be the primary focus of that domain concern. However, that architect **will fail for many reasons**. **First, it doesn’t matter how fast the system is if it isn’t available when needed. Second, as the domain grows the system must be able to also scale to finish end-of-day processing in time. Third, the system must also be reliable so that it doesn’t crash as end-of-day fund prices are being calculated. Fourth, what if the system crashes in the middle of processing? It must be able to recover and restart where the pricing left off**. **Finally, are the fund prices being calculated correctly?** So, in addition to performance, **the architect must also equally place a focus on availability, scalability, reliability, recoverability, and auditability.**

Extracting Architecture Characteristics from Requirements

Some architecture characteristics come from explicit statements in requirements documents. For example, explicit expected numbers of users and scale commonly appear in domain or domain concerns.
Others implicitly come from inherent domain knowledge by architects, one of the many reasons that domain knowledge is always beneficial for architects.

Furthermore, architects should **adopt a curious approach, actively seeking more detailed requirements and clarifying assumptions**. By asking targeted questions and probing for deeper insights, **architects can uncover hidden requirements and better understand the nuances of the domain**.

For example, suppose an architect designs an application that handles class registration for university students. To make the math easy, assume that the school has 1,000 students and 10 hours for registration. Should an architect design a system **assuming consistent scale**, making the **implicit assumption** that the students during the registration process will distribute themselves evenly over time? Or, based on knowledge of university students' habits and proclivities, should the architect design a system that can handle all 1,000 students attempting to register in the last 10 minutes? Anyone who understands how much students stereotypically procrastinate knows the answer to this question! **Rarely will details like this appear in requirements documents, yet they do inform the design decisions**.

How are you supposed **to be a great architect** if you only get the chance to architect fewer than a half dozen times in your career?
A few years ago, Ted Neward, a well-known architect, devised **architecture katas**, a clever method to allow nascent architects a way **to practice deriving architecture characteristics from domain-targeted descriptions**.

Check Neal Ford blog for Architecture Katas with additional context : https://nealford.com/katas

Case Study: An Architecture Kata

This Study Case of an architecture kata “Silicon Sandwiches” is to show how architects derive architecture characteristics from requirements.

Description

- A national sandwich shop wants to enable online ordering (in addition to its current call-in service).

Users

- Thousands, perhaps one day millions

Requirements

- Users will place their order, then be given a time to pick up their sandwich and directions to the shop (which must integrate with several external mapping services that include traffic information)
- If the shop offers a delivery service, dispatch the driver with the sandwich to the user
- Mobile-device accessibility
- Offer national daily promotions/specials
- Offer local daily promotions/specials
- Accept payment online, in person, or upon delivery

Additional context

- Sandwich shops are franchised, each with a different owner
- Parent company has near-future plans to expand overseas
- Corporate goal is to hire inexpensive labor to maximize profit

Given this scenario, how would an architect derive architecture characteristics? Each part of the requirement might contribute to one or more aspects of architecture (and many will not).
Now, let's use what we have learned about characteristics criteria and identifying the characteristics.

First, separate the candidate architecture characteristics into explicit and implicit characteristics.

Explicit Characteristics

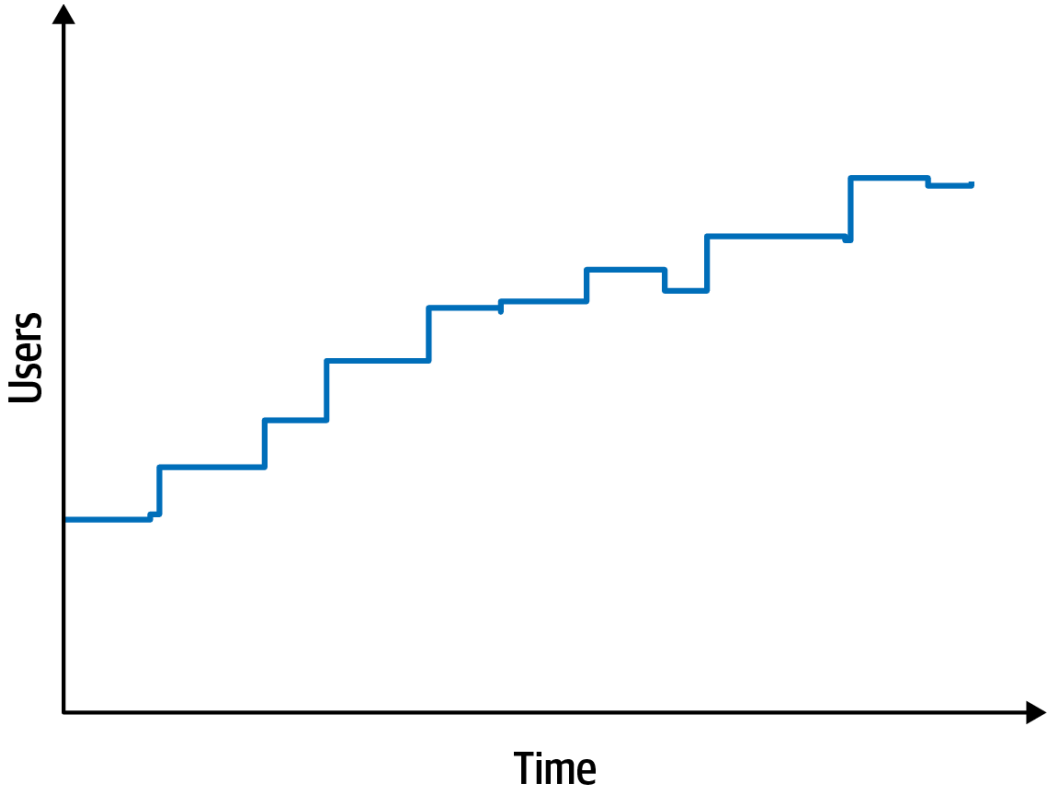
Explicit architecture characteristics appear in a requirements specification as part of the necessary design. For example, a shopping website may aspire to support a particular number of concurrent users, which domain analysts specify in the requirements.
An architect should consider **domain-level predictions** about expected metrics.

One of the first details that should **catch an architect's eye is the number of users**: currently thousands, perhaps one day millions (this is a very ambitious sandwich shop!). Thus, **scalability**—the ability to handle a large number of concurrent users without serious performance degradation—**is one of the top architecture characteristics**. Notice that the problem statement **didn't explicitly ask for scalability**.
However, we also probably need **elasticity**—the ability to handle bursts of requests. These two characteristics often appear lumped together, but they have different constraints.

Question: What is the difference between scalability and elasticity?

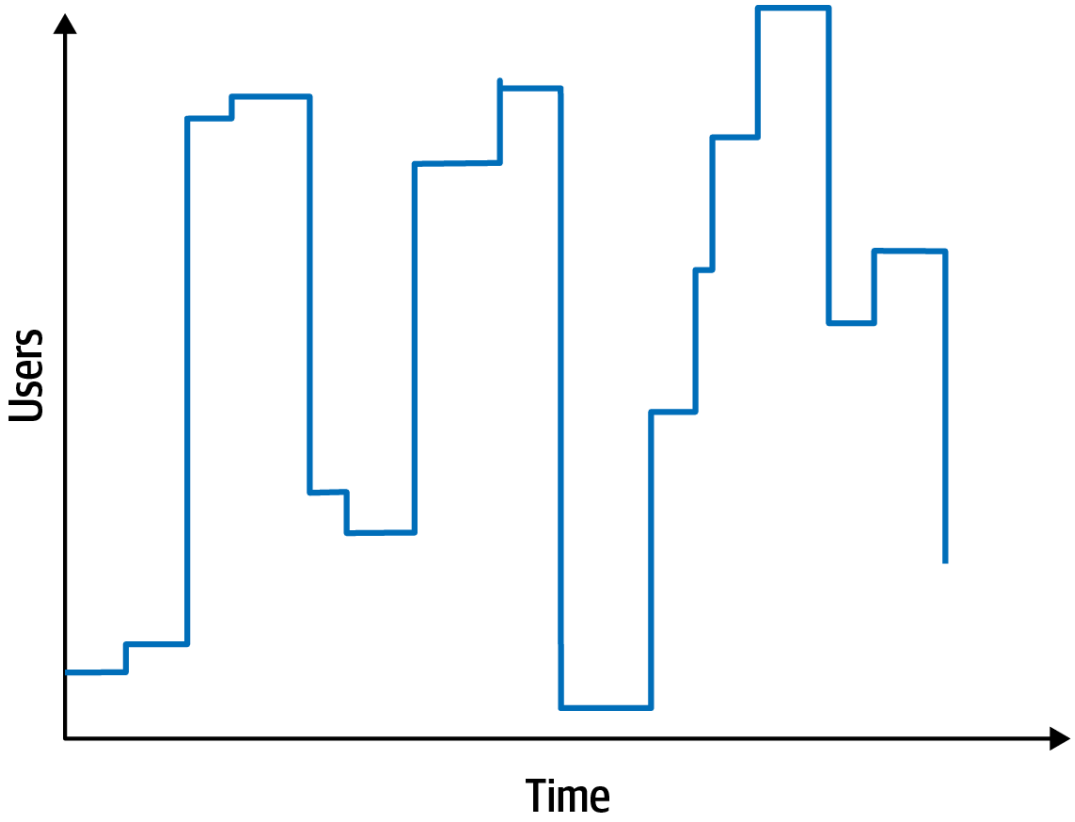
Scalability is the ability of a system to **handle increased load** **(which is often predicted)** by manually adding more **resources** either vertically (enhancing existing systems) or horizontally (adding more systems) or using existing **resources** more **efficiently**. It's geared towards **long-term growth**.

Figure : Scalability measures the performance of concurrent users



Elasticity, on the other hand, responds to **short-term**, is the ability of a system to **dynamically scale up and down based on the current demand (which is somehow unpredictable) and without human intervention**, often realized through **cloud-based infrastructures**, aiming to match the **exact resource requirement at any moment**.

Figure : Elastic systems must withstand bursts of users



Some systems are scalable but not elastic. For example, consider a hotel reservation system. Absent special sales or events, **the number of users is probably consistent and expected**. In contrast, consider a concert ticket booking system. As new tickets go on sale, fervent fans will **flood the site with unexpected growth in the number of users, requiring high degrees of elasticity.**
Elasticity **adds a layer of intelligence and automation to scalability**, allowing systems to respond to immediate changes in demand without manual intervention. **This distinction is crucial in cloud computing, where the pay-as-you-go pricing model makes elasticity economically attractive.**
Often, elastic systems also inherently require scalability as a foundation for adding or removing resources.

The requirement for **elasticity did not appear** in the Silicon Sandwiches **requirements, yet the architect should identify this as an important consideration.** Consider a sandwich shop. Is its traffic consistent throughout the day? Or does it endure bursts of traffic around mealtimes? Almost certainly the latter. **Thus, a good architect should identify this potential architecture characteristic.**

An architect should consider each of these business requirements in turn to see if architecture characteristics exist:

- Users will place their order, then be given a time to pick up their sandwich and directions to the shop (which must provide the option to integrate with external mapping services that include traffic information).

  External mapping services **imply integration points**, which may **impact** aspects such as **reliability**. For example, if a developer builds a system that relies on a third-party system, yet calling it fails, it impacts the reliability of the calling system. However, architects must also be **wary of over-specifying architecture characteristics**.

- If the shop offers a delivery service, dispatch the driver with the sandwich to the user.

  No special architecture characteristics seem necessary to support this requirement.

- Mobile-device accessibility.

  This requirement will **primarily affect the design of the application**, **pointing toward building either a portable web application or several native web applications.** Given the budget constraints and simplicity of the application, **the design points toward a mobile-optimized web application.** Thus, the architect may want to define some **specific performance architecture characteristics** for **page load time and other mobile-sensitive characteristics**. Notice that **the architect** shouldn't act alone in situations like this, but should instead **collaborate with user experience designers, domain stakeholders, and other interested parties to vet decisions like this.**

- Offer national daily promotions/specials.
- Offer local daily promotions/specials.

  Both of these requirements **specify customizability** across both promotions and specials. Notice that requirement one also implies **customized traffic information based on address**. Based on all three of these requirements, the architect may consider **customizability as an architecture characteristic**. For example, an architecture style such as microkernel architecture supports **customized behavior** extremely well by defining a plug-in architecture. In this case, the **default behavior** appears in **the core**, and developers write the **optional**

customized parts, based on location, via plug-ins. However, a **traditional design** can also accommodate this requirement **via design patterns** (such as Template Method). This is common in architecture and requires architects to **constantly weigh trade-offs** between competing options. *Recap on Section "Design Versus Architecture and Trade-Offs"*.

- Accept payment online, in person, or upon delivery.

  Online payments imply **security**, but **nothing** in this requirement suggests a **special** heightened **level of security** beyond what's implicit.

- Sandwich shops are franchised, each with a different owner.

  This requirement may impose cost restrictions on the architecture—the architect should **check the feasibility** (applying constraints like cost, time, and staff skill set) to see if a simple or sacrificial architecture is warranted.

- The parent company has near-future plans to expand overseas.

  This requirement implies **internationalization**, or i18n. Many design techniques exist to handle this requirement, which **shouldn't require special structure** to accommodate.

- The corporate goal is to hire inexpensive labor to maximize profit.

  This requirement suggests that **usability** will be important, but again is **more concerned with design than architecture characteristics (No need to consider it)**.

The third architectural characteristic we derive from the preceding requirements is **performance**: no one wants to buy from a sandwich shop that has poor performance, especially at peak times. However, **performance is a nuanced concept**—what kind of performance should the architect design for? *Check the Next Section about performance*.

We also want to define **performance** numbers in **conjunction** with **scalability** numbers. In other words, we must establish a **baseline of performance without particular scale**, as well as determine what an **acceptable level of performance is given a certain number of users**. **Quite often, architecture characteristics interact with one another, forcing architects to define them in relation to one another.**

IMPLICIT CHARACTERISTICS

Many architecture characteristics aren't specified in requirements documents, yet they make up an important aspect of the design.
One implicit architecture characteristic the system might want to support is **availability**: making sure users can access the sandwich site. Closely related to availability is **reliability**: making sure the site stays up during interactions—no one wants to purchase from a site that continues dropping connections.

Security appears as an **implicit** characteristic in every system: no one wants to create insecure software. However, it may be prioritized depending on criticality.
An architect might **assume that payments** should be handled by a **third party**. Thus, as long as developers follow general security hygiene (not passing credit card numbers as plain text, not storing too much information, and so on), the architect **shouldn't need any special structural design** to accommodate **security**; good design in the application will suffice.

DESIGN VERSUS ARCHITECTURE AND TRADE-OFFS

The last **major architecture characteristic** from the **requirements**: **customizability**. Notice that several parts of the problem domain offer **custom behavior**: recipes, local sales, and directions that may be locally overridden. **This design element isn't critical to the success of the application though**.

An architect would likely prioritize **customizability**, but the question then becomes: **architecture or design?**, note that **design resides within the architecture**. In the customizability case of Silicon Sandwiches, **the architect could choose an architecture style like microkernel for customization.** However, if the architect chose another style because of **competing concerns**, **developers could implement the customization using the Template Method design pattern.**

Which design is better?

Like in all architecture, **it depends** on a number of factors. First, **are there good reasons**, such as performance and coupling, **not to implement a microkernel architecture?** Second, are **other desirable architecture characteristics more difficult** in one design versus the other? Third, how much would it **cost** to **support all** the architecture **characteristics in each design versus pattern**? **This type of architectural trade-off analysis makes up an important part of an architect's role.**

Above all, it is critical for the architect to collaborate with the developers, project manager, operations team, and other co-constructors of the software system. No architecture decision should be made isolated from the implementation team (which

leads to the dreaded Ivory Tower Architect anti-pattern). In the case of Silicon Sandwiches, the architect, tech lead, developers, and domain analysts should collaborate to decide how best to implement customizability.

It is important to note that there are no correct answers in choosing architecture characteristics, only incorrect ones (or, as Mark notes in one of his well-known quotes):

"There are no wrong answers in architecture, only expensive ones."
And in other words, there is no best design in architecture, only a least worst collection of trade-offs.

Architects shouldn't stress too much about discovering the exactly correct set of architecture characteristics—developers can implement functionality in a variety of ways. However, correctly identifying important structural elements may facilitate a simpler or more elegant design.

Architects must also **prioritize these architecture characteristics** toward trying to find the simplest required sets. **A useful exercise** once the team has made a first pass at identifying the architecture characteristics is to **try to determine the least important one—if you must eliminate one, which would it be?** Generally, architects are more likely to **cull the explicit architecture characteristics**, as many of the **implicit ones support general success**. By attempting to **determine the least applicable one**, architects can help **determine critical necessity**.

In the case of Silicon Sandwiches, which architecture characteristic is least important? Again, no absolute correct answer exists. However, in this case, **the solution could lose either customizability or performance**. We could eliminate customizability as an architecture characteristic and plan to implement that behavior as part of application design. Of the operational architecture characteristics, **performance is likely the least critical for success**. Of course, the developers don't mean to build an application that has terrible performance, but rather one that **doesn't prioritize performance over other characteristics, such as scalability or availability.**

Measuring and Governing Characteristics

Architects must deal with the extraordinarily wide variety of architecture characteristics. Operational aspects like performance, elasticity, and scalability coming with structural concerns such as modularity and deployability.

Measuring Architecture Characteristics

Operational Measures

THE MANY FLAVORS OF PERFORMANCE

Performance exhibits **remarkable depth and variety**, **extending beyond general metrics like request-response cycles**. Architects and DevOps engineers have developed detailed performance budgets to address specific application areas, such as **optimizing first-page render times to a target of 500 ms to enhance user engagement**. Furthermore, progressive organizations have introduced **K-weight budgets for page downloads, setting limits on the bytes' worth of libraries and frameworks** to accommodate physical **network constraints**, **particularly for mobile devices** in areas with high latency. **This nuanced approach to performance**, **tailored to the unique demands** of modern web and mobile applications, illustrates the intricate measures organizations adopt **to refine user experience and operational efficiency.**

Structural Measures

Some objective measures are not so obvious *as performance*. **What about internal structural characteristics**, such as **well-defined modularity**? **Some metrics and common tools do allow architects to address some critical aspects of code structure**, albeit along narrow dimensions.

An obvious measurable aspect of code is complexity, defined by **the cyclomatic complexity metric**.

CYCLOMATIC COMPLEXITY

Cyclomatic Complexity (CC) is a **code-level metric** designed to provide an **objective measure for the complexity of code**, **at the function/method, class, or application level**, developed by Thomas McCabe, Sr., in 1976.

It is computed by applying graph theory to code, **specifically decision points**, which cause different execution paths. For example, if a function has no decision statements (such as if statements), then CC = 1. If the function had a single conditional, then CC = 2 because two possible execution paths exist.

The formula for calculating the CC for a single function or method is ***CC = E - N + 2***, where **N** represents nodes (points of decision), and **E** represents edges (possible decisions).

```
public void decision(int c1, int c2) {
```

```
    if (c1 < 100) // point of decision

        return 0;// possible decision

    else if (c1 + C2 > 500)// point of decision

       return 1;// possible decision

    else

      return -1;// possible decision

}
```

The cyclomatic complexity for Example is **3** (=3 – 2 + 2)

WHAT'S A GOOD VALUE FOR CYCLOMATIC COMPLEXITY?

Of course, like all answers in software architecture: **it depends!** It depends on the complexity of the problem domain.
For example, if you have an algorithmically complex problem, the solution will yield complex functions. Some of the key aspects of CC for architects to monitor: **are functions complex because of the problem domain or because of poor coding?**

In general, the industry thresholds for CC suggest that a value under 10 is acceptable, barring other considerations such as complex domains.

Engineering practices like test-driven development have the accidental (but positive) side effect of generating smaller, less complex methods on average for a given problem domain. When practicing TDD, developers try to write a simple test, then write the smallest amount of code to pass the test. This focus on discrete behavior and good test boundaries encourages **well-factored**, **highly cohesive** **methods that exhibit low CC.**

Tools for Code Analysis

Various tools for code analysis exist that can help architects and developers evaluate different aspects of code quality. Tools like those available in Visual Studio provide insights into **code maintainability, cyclomatic complexity, potential bugs, and adherence to coding standards.** These tools, while not offering a comprehensive measure of internal code quality, enable the identification and mitigation of critical structural issues. For example, static analysis

tools can detect code smells, duplicated code, and overly complex methods, guiding refactoring efforts. By integrating these tools into the development workflow, teams can continuously monitor and improve code quality, addressing some of the critical aspects of code structure mentioned earlier, albeit along narrow dimensions. This approach, although limited, forms a part of a broader strategy to maintain high code quality.

Governing Architecture Characteristics

Once architects have established architecture characteristics and prioritized them, how can they make sure that developers will respect those priorities? Modularity is a great example of an aspect of architecture that is important but not urgent; on many software projects, urgency dominates, yet architects still need a mechanism for governance.

Governance is an important responsibility of the architect role. As the name implies, **the scope of architecture governance covers any aspect of the software development process that architects want to exert an influence upon**. For example, ensuring software quality within an organization falls under the heading of architectural governance **because it falls within the scope of architecture, and negligence can lead to disastrous quality problems**.

Fortunately, increasingly sophisticated solutions exist to relieve this problem from architects. The drive toward **automation** on software projects **spawned by Extreme Programming created continuous integration**, **which led to further automation into operations**, which **we now call DevOps**, continuing through to **architectural governance**. The book Building Evolutionary Architectures (O'Reilly) describes a family of techniques, called **fitness functions, used to automate many aspects of architecture governance**.

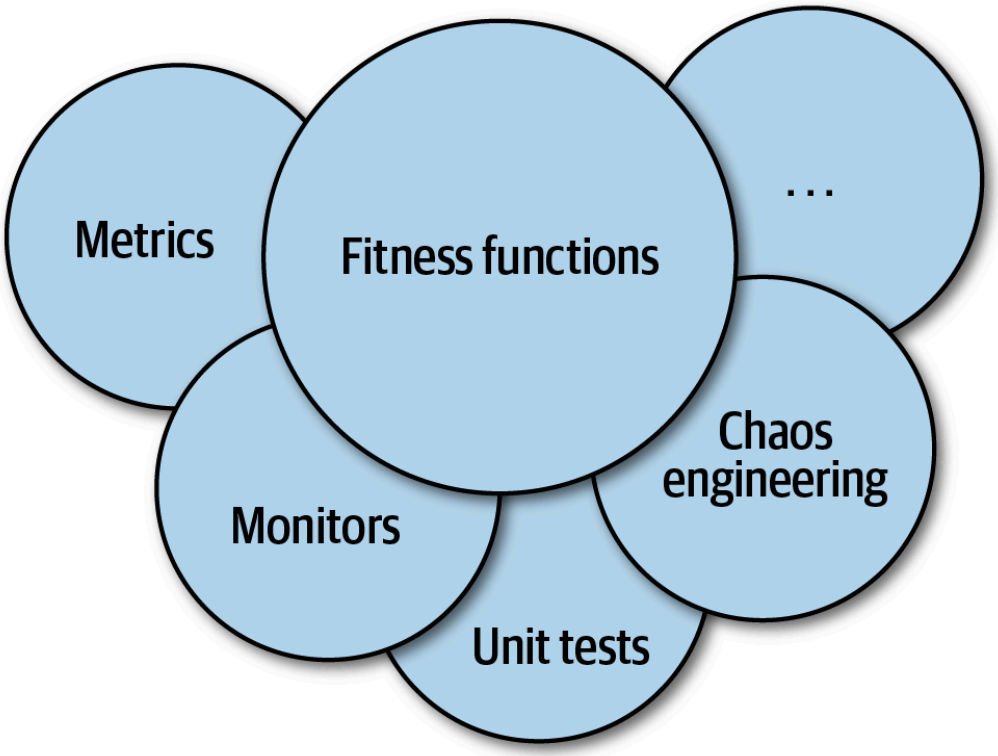
Fitness Functions

An architect may want to provide an objective **measure** indicating the **quality of the outcome**. That guidance mechanism is called an **Architecture Fitness Function** :

"Any mechanism that provides an objective integrity assessment of
some architecture characteristic or combination of architecture characteristics"
In other words, an object function is used to assess
how close the output comes to achieving the aim.

For example, suppose a developer needed to solve the traveling salesperson problem, a famous problem used as a basis for machine learning. Given a salesperson and a list of cities they must visit, with distances between them, **what is the optimum route?** If a developer designs a genetic algorithm to solve this problem, **one fitness function might evaluate the length** of the route, as the **shortest possible** one represents **highest success**. Another fitness function might be to **evaluate** the **overall cost** associated with the route and attempt to **keep cost at a minimum**.

Fitness functions are not some new framework for architects to download, but rather a **new perspective** on many existing tools such as metrics, monitors, unit testing libraries, chaos engineering, and so on.



For example, in “Coupling” there are metrics to allow architects to assess modularity. Here are a couple of examples of fitness functions that **test various aspects of modularity.**

Cyclic dependencies

Modularity is an **implicit architecture characteristic** that most architects care about, **because poorly maintained modularity harms the structure of a code base**; **thus, architects should**

place a high priority on maintaining good modularity. However, **forces work against the architect's** good intentions on many platforms. For example, when coding in any popular Java or .NET development environment, most programmers develop **the habit of swatting the auto-import dialog away** like a reflex action. However, **arbitrarily importing classes or components between one another spells disaster for modularity.**

How can architects govern this behavior without constantly looking over the shoulders of developers? Code reviews help but happen too late.

The solution to this problem is to write a fitness function to look after cycles, as shown in the next example.

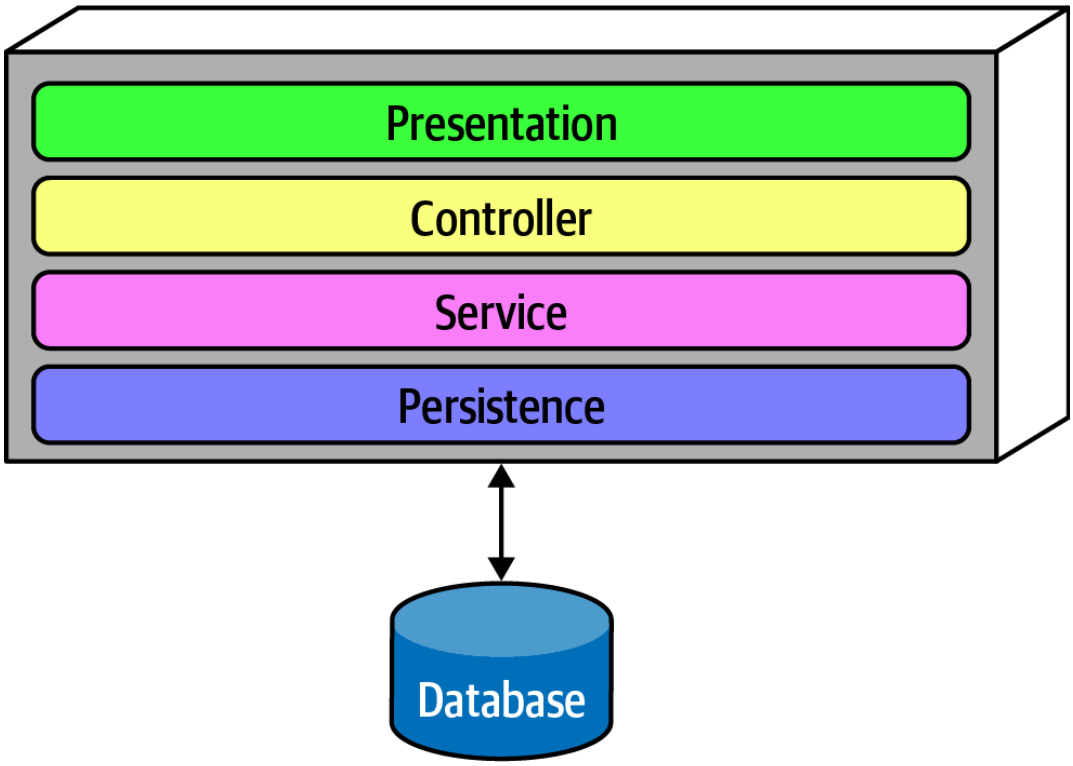
```
using NUnit.Framework;
using System.Collections.Generic;
using NetDepend;

namespace MyProject.Tests
{
    [TestFixture]
    public class CycleTest
    {
        private NetDepend netDepend;

        [SetUp]
        public void Init()
        {
            netDepend = new NetDepend();

netDepend.AddDirectory(@"\path\to\project\persistence\classes");
            netDepend.AddDirectory(@"\path\to\project\web\classes");
            netDepend.AddDirectory(@"\path\to\project\thirdpartyjars");
        }

        [Test]
        public void TestAllPackages()
        {
            var packages = netDepend.Analyze();
            Assert.IsFalse(netDepend.ContainsCycles(), "Cycles exist");
        }
    }
```

```
}
```

Note: Architects must ensure that developers understand the purpose of the fitness function before imposing it on them.

How can architects ensure that developers **adhere to the intended layered architecture?**

One such tool is **ArchUnit**, a Java testing framework. ArchUnit provides a variety of **predefined governance rules** codified as unit tests and allows architects to write specific tests that address **modularity**.
Consider the layered architecture illustrated in the figure:



When designing a layered monolith, the architect defines the layers for good reason (motivations, trade-offs, and other aspects of the layered architecture). However, **how can the architect ensure that developers will respect those layers?** Some developers may not understand the importance of the patterns. But **allowing implementers to erode the reasons for the architecture hurts the long-term health of the architecture.**

ArchUnit allows architects to address this problem via a fitness function by defining the desirable relationship between layers and writing a verification fitness function to govern it, shown in Example.

Example: ArchUnit fitness function to govern layers

```
layeredArchitecture()
    .layer("Controller").definedBy("..controller..")
    .layer("Service").definedBy("..service..")
    .layer("Persistence").definedBy("..persistence..")

    .whereLayer("Controller").mayNotBeAccessedByAnyLayer()
    .whereLayer("Service").mayOnlyBeAccessedByLayers("Controller")
    .whereLayer("Persistence").mayOnlyBeAccessedByLayers("Service");
```

A similar tool in the .NET space, **ArchUnitNET** and **NetArchTest**, allows similar tests for that platform; a layer verification in C# appears in the following example.

Example: NetArchTest for layer dependencies

```
// Classes in the presentation should not directly reference repositories
var result = Types.InCurrentDomain()
    .That()
    .ResideInNamespace("NetArchTest.SampleLibrary.Presentation")
    .ShouldNot()
    .HaveDependencyOn("NetArchTest.SampleLibrary.Data")
    .GetResult()
    .IsSuccessful;
```

The influential book The Checklist Manifesto by Atul Gawande (Picador) described **how professions** such as airline pilots and surgeons **use checklists**. It's not because those professionals don't know their jobs or are forgetful. Rather, **when professionals do a highly detailed job over and over, it becomes easy for details to slip by**; ***a short checklist forms an effective reminder***. This is the **correct perspective on fitness functions**—rather than a heavyweight governance mechanism, fitness functions provide a mechanism for architects to express **important architectural principles** and **automatically verify them**.

Scope of Architecture Characteristics

In “**Structural Measures"**, we discuss **code-level metrics such as cyclomatic complexity**. However, all these metrics only **reveal low-level details** about the code, and cannot evaluate **dependent components** (such as **databases**) that still **impact many architecture characteristics**, especially **operational ones**. For example, no matter how much an architect puts effort into designing a performant or elastic code base, **if the system uses a database that doesn’t match those characteristics, the application won’t be successful.**
Thus, architects need another method to measure these kinds of dependencies. That led to defining the term **architecture quantum**. To understand the architecture quantum definition, we must preview one key metric here, **connascence**.

Connascence

Many of the code-level coupling metrics, such as **afferent and efferent coupling**, reveal **details at a too fine-grained level** for architectural analysis.
Connascence:

> ***Two components are connascent if a change in one would require the other to be modified in order to maintain the overall correctness of the system***

They defined two types of **connascence**: **static**, discoverable via **static code** analysis, and **dynamic**, concerning **runtime behavior**. **To define the architecture quantum, we needed a measure of how components are “wired” together**, which corresponds to the **connascence** concept. For example, if two services in a microservices architecture share the same class definition of some class, like address, we say they are **statically connascent** with each other—**changing the shared class** requires **changes to both services**.
For **dynamic connascence**, we define two types: **synchronous and asynchronous**. **Synchronous** calls between two distributed services have the **caller wait for the response from the callee**. On the other hand, asynchronous **calls allow fire-and-forget** semantics in event-driven architectures, allowing two different services to differ in operational architecture.

Architecture Quantum

An independently deployable artifact with high functional cohesion and synchronous connascence.

This definition contains several parts, dissected here:

Independently deployable:
An architecture quantum includes all the necessary components to **function independently from other parts** of the architecture. For example, if an application uses a database, **it is part**

of the quantum because the system **won't function without it**. However, in the microservices architecture style, each service includes its own database, creating **multiple quanta** within that architecture.

High functional cohesion:
Cohesion in component design refers to how well the contained code is **unified in purpose**. In **microservices** architectures, developers typically **design each service to match a single workflow** (a bounded context), thus exhibiting **high functional cohesion**.

Synchronous connascence:
Implies **synchronous calls** within an **application context** or **between distributed services that form an architecture quantum**. For example, if one service in a microservices architecture calls another one synchronously, and the caller is much more scalable than the callee, timeouts and other reliability concerns will occur. Thus, **synchronous calls create dynamic connascence for the length of the call—if one is waiting for the other, their operational architecture characteristics (such as scalability and elasticity) must be the same for the duration of the call.**

For another example, consider a microservices architecture with a Payment service and an Auction service. When an auction ends, the **Auction service sends payment information to the Payment service**. However, let's say that the payment service can only **handle a payment every 500 ms**—what happens when **a large number of auctions end at once**? A poorly designed architecture would allow the first call to go through and allow **the others to time out**. Alternatively, an architect might **design an asynchronous communication link** between Payment and Auction, allowing the message **queue to temporarily buffer differences**. In this case, **asynchronous connascence creates a more flexible architecture**.

Bounded Context (of DDD)

DDD defines the bounded context, where everything related to the domain is visible internally but opaque to other bounded contexts. **Before DDD, developers creating common shared artifacts caused** a **host of problems, such as coupling**, **more difficult coordination**, **and increased complexity**. **The bounded context concept recognizes that each entity works best within a localized context**. Thus, instead, each problem domain can create its own and **reconcile differences at integration points**.

The architecture quantum concept provides the **new scope for architecture characteristics**. In modern systems, architects define architecture **characteristics at the quantum level** rather than system level. By looking at a narrower scope for important **operational concerns**, architects may **identify** architectural **challenges early**, leading to hybrid architectures. To illustrate scoping provided by the architecture quantum measure, consider another architecture kata, Going, Going, Gone.

Case Study: Going, Going, Gone

Description:
An auction company wants to take its auctions online to a nationwide scale. Customers choose the auction to participate in, wait until the auction begins, then bid as if they are there in the room with the auctioneer.

Users:
Scale up to hundreds of participants per auction, potentially up to thousands of participants, and as many simultaneous auctions as possible.

Requirements:
- Auctions must be as real-time as possible.
- Bidders register with a credit card; the system automatically charges the card if the bidder wins.
- Participants must be tracked via a reputation index.
- Bidders can see a live video stream of the auction and all bids as they occur.
- Both online and live bids must be received in the order in which they are placed.

Additional context:
- The auction company is expanding aggressively by merging with smaller competitors.
- The budget is not constrained. This is a strategic direction.
- The company just exited a lawsuit where it settled a suit alleging fraud.

An architect must consider each of these requirements to ascertain architecture characteristics:

1. “Nationwide scale,” “scale up to hundreds of participants per auction, potentially up to thousands of participants, and as many simultaneous auctions as possible,” “auctions must be as real-time as possible.”
   Each of these requirements implies both scalability and elasticity. While the requirements explicitly call out scalability, **elasticity represents implicit characteristics**. When considering auctions, do users all politely spread themselves out during the course of bidding, or do they become more frantic near the end? Domain knowledge is crucial for architects to pick up implicit architecture characteristics. Given the real-time nature of auctions, an architect will certainly consider performance a key architecture characteristic.

2. “Bidders register with a credit card; the system automatically charges the card if the bidder wins,” “company just exited a lawsuit where it settled a suit alleging fraud.”

Both these requirements clearly point to **security** which is an implicit architecture characteristic in virtually every application. **Should an architect design something special to accommodate security, or will general design and coding hygiene suffice?** For example, as long as developers make sure not to store credit card numbers in plain text, to encrypt while in transit, and so on, then the architect shouldn't have to build special considerations for security.
However, the second phrase should make **an architect pause and ask for further clarification**. Clearly, some aspect of security **(fraud)** was a problem in the past, thus the architect should ask for further input no matter what level of security they design.

3. "Participants must be tracked via a reputation index."
   The track part of this requirement might suggest characteristics such as **auditability and loggability**. The deciding factor again goes back to the defining characteristic—**is this outside the scope of the problem domain?** *Architects must remember that yielding architecture characteristics represents only a small part of the overall effort to design and implement an application! Architects look for* ***requirements with structural impact*** *not already covered by the domain.*
   The phrase "reputation index" **isn't a standard definition** like more common architecture characteristics. As a counter example, when architects discuss elasticity, they can talk about the architecture characteristic purely in the **abstract—it doesn't matter what kind of application they consider**: banking, catalog site, streaming video, and so on.

4. "Auction company is expanding aggressively by merging with smaller competitors."

   This requirement may not have an immediate impact on application design, it might become the determining factor in a trade-off like choosing details such as communication protocols for integration architecture if integration with newly merged companies is a concern.

5. "The budget is not constrained. This is a strategic direction.".
   Some architecture katas impose budget restrictions on the solution to represent a common real-world trade-off. However, in the Going, Going, Gone kata, it does not.

6. "Bidders can see a live video stream of the auction and all bids as they occur," "both online and live bids must be received in the order in which they are placed."
   This requirement presents an **interesting architectural challenge**, definitely **impacting the structure** of the application. Is the availability of the one bidder more important than availability for one of the hundreds of bidders? Obviously, **one is clearly more critical: if the auctioneer cannot access the site, online bids cannot occur for anyone. Reliability commonly appears with availability**; it addresses operational aspects such as **uptime**, as well as **data integrity** and other measures of how **reliable** an application is. For example, in an auction site, the architect must ensure that the **message ordering is reliably correct**.

This last requirement highlights the **need** for using the **architecture quantum measure** as architects scope architecture **characteristics at the quantum level**. For example, in Going, Going, Gone, an architect would **notice that different parts** of this architecture **need different characteristics**: **streaming bids, online bidders, and the auctioneer** are three obvious choices. The architecture quantum measure is a way to think about **deployment, coupling, where data should reside, and communication styles within architectures.** In this kata, an architect can analyze the differing architecture **characteristics per architecture quantum**, leading to hybrid architecture design earlier in the process.

Thus, for Going, Going, Gone, we identified the following quanta and corresponding architecture characteristics:

Bidder feedback
Encompasses the bid stream and video stream of bids :

- Availability
- Scalability
- Performance

Auctioneer
The live auctioneer :

- Availability
- Reliability
- Scalability
- Elasticity
- Performance
- Security

Bidder
Online bidders and bidding :

- Availability

- Reliability
- Scalability
- Elasticity

Component-Based Thinking

In the “Modularity” chapter, we discussed modules as a collection of related code. However, architects typically think in terms of **components**, **the physical manifestation of a module**. Developers physically package modules in different ways as we call components such as DLL files in .Net or JAR files in Java.
So we will discuss architectural considerations around components, ranging from scope to discovery.

Component Scope

The simplest component is a **wrapper** that is often called a **library** which represents a **higher level of modularity**. It tends to run in the **same memory address** as the calling code and **communicate via language function call** mechanisms.
Components also appear as **subsystems or layers** in architecture.
Libraries are usually compile-time dependencies.
Another type of **component**, **a service**, tends to **run in its own address space** and **communicates via low-level networking protocols** like TCP/IP or **higher-level formats** like REST or **message queues**, **forming stand-alone**, deployable units in architectures **like microservices**.
Nothing requires an architect to use components—it’s often useful to have a higher level of modularity than the lowest level.
Components, as the f**undamental modular building blocks in architecture**, are a critical consideration for architects, particularly in **making decisions** regarding the **top-level partitioning** within the architecture.

Architect Role

Typically, **the architect defines, refines, manages, and governs components within an architecture**. Software architects, **in collaboration** with business analysts, subject matter experts, developers, QA engineers, operations, and enterprise architects, **create the initial design** for software, incorporating the architecture **characteristics** and the **requirements** for the software system.

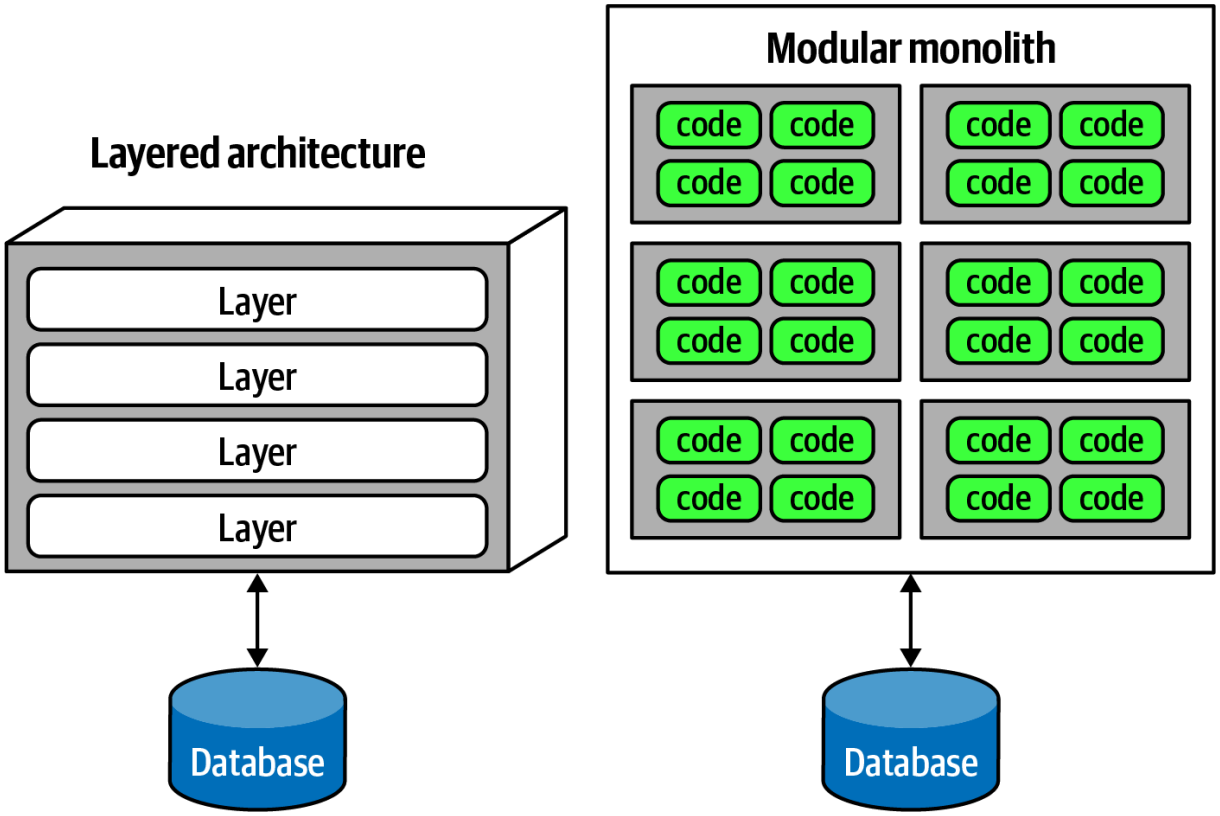
Virtually, architecture is independent from the development process. The primary exception to this rule entails the engineering practices pioneered in the **various flavors of Agile** software development, **particularly in the areas of deployment and automating governance**. Thus, architects ultimately don't care where requirements originate: waterfall-style analysis and design, Agile story cards...

Components consist of classes or functions, whose **design falls under the responsibility of tech leads or developers.**
It's not that architects shouldn't involve themselves in class design (particularly applying design patterns), they should **avoid micromanaging each decision** from top to bottom in the system. **If architects never allow other roles to make decisions of consequence, the organization will struggle with empowering the next generation of architects.**

An architect must identify components as <u>one of the first tasks on a new project</u>. But before an architect can identify components, they must know how to partition the architecture.

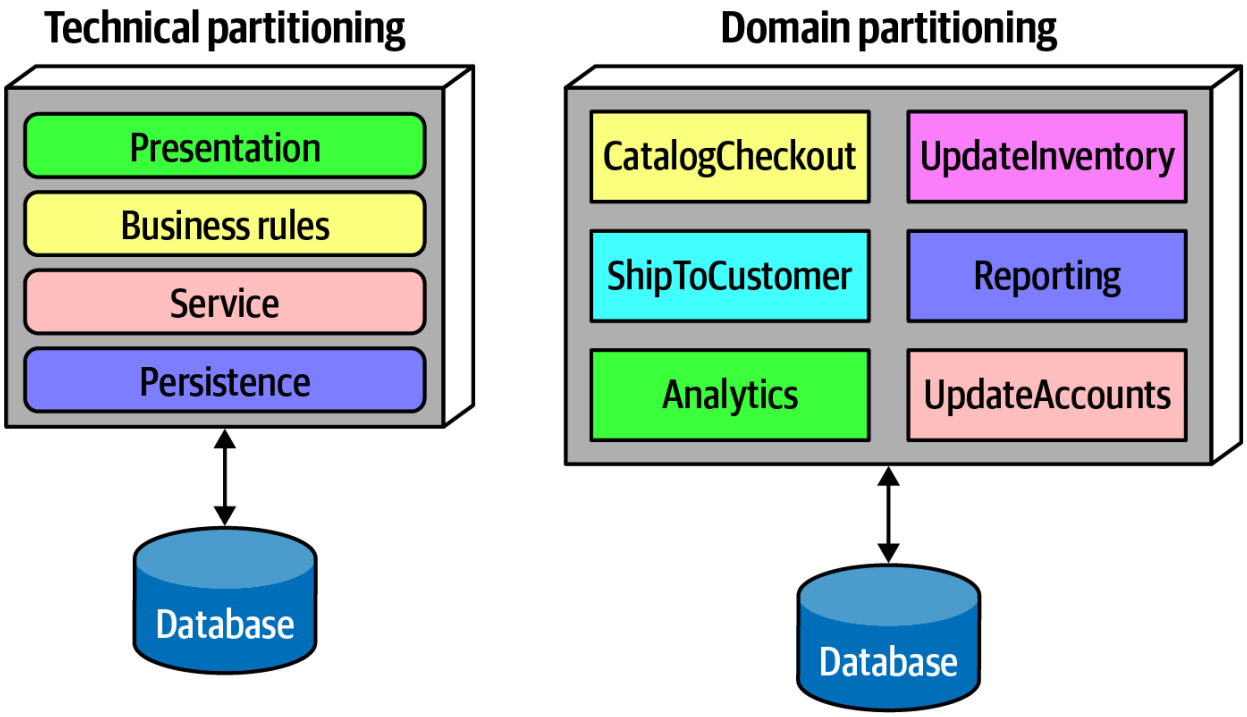
Architecture Partitioning

Because components represent a general containership mechanism, an architect can build any type of partitioning they want. Several common styles exist, with different sets of trade-offs. Here we discuss an important aspect of architecture styles, the top-level partitioning in an architecture.



In the Figure, The **modular monolith**, a single deployment unit associated with a database and **partitioned around domains** rather than technical capabilities. The modular monolith and layered architecture are two styles that represent **different ways to top-level partition the architecture**.
The top-level partitioning defines the fundamental architecture style and way of partitioning code.

Organizing architecture based on technical capabilities like the layered monolith represents technical top-level partitioning. Consider the following Figure.



The basic concept of layered architecture predates it by decades, the Model-View-Controller design pattern matches with this architectural pattern, making it easy for developers to understand. Thus, it is often the default architecture in many organizations.

An interesting side effect of the predominance of the layered architecture is that when using a layered architecture, it makes some sense to have all the backend developers sit together in one department, the DBAs in another, the presentation team in another, and so on. **Because of Conway's law, this makes some sense in those organizations.**

Conways' Law

> ***Organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations.***

This law suggests that when a group of people designs some technical artifact, the communication structures between the people end up replicated in the design. People at all levels of organizations see this law in action, and they sometimes make decisions based on it.

A well-designed architecture aligns with the organizational structure, promoting clearer communication, better-defined roles, and more efficient collaboration and resource allocation.

Domain Partitioning

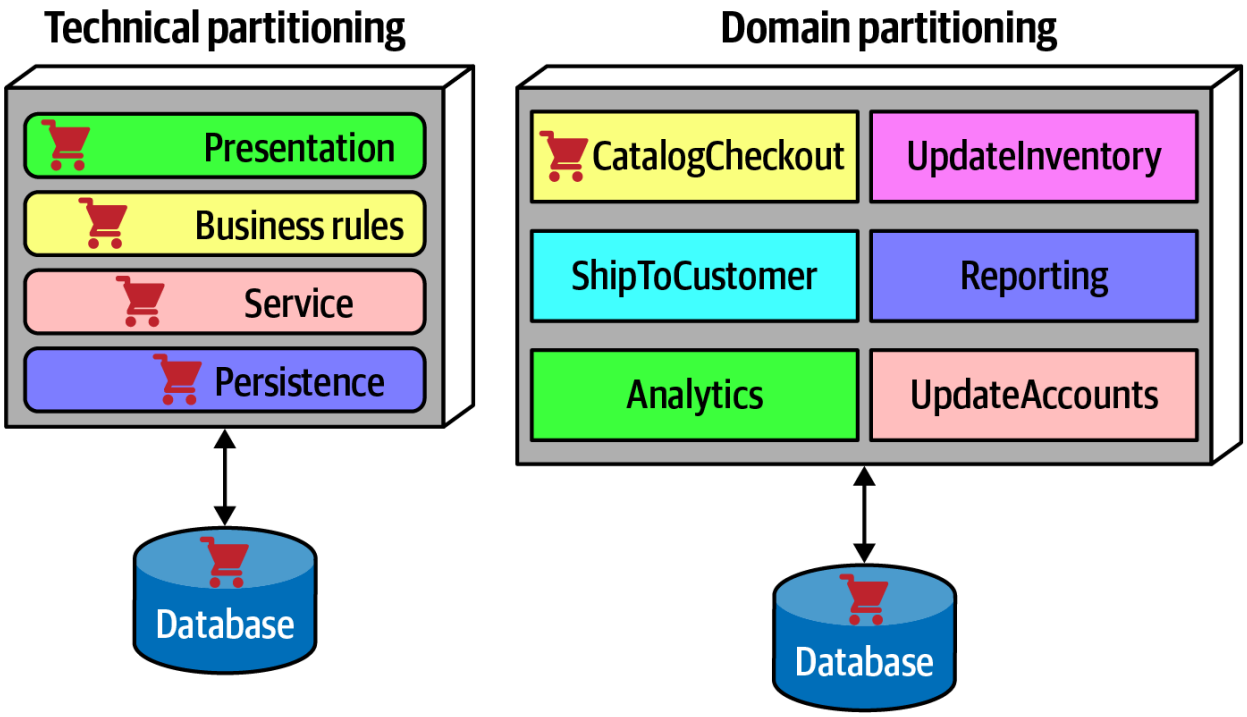
The other architectural variation in the previous Figure represents domain partitioning, **inspired by** the Eric Evan book **Domain-Driven Design**. In DDD, the architect identifies domains or workflows independent and decoupled from each other. The **microservices architecture style is based on this philosophy**. In a **modular monolith**, the architect **partitions** the architecture **around domains** or workflows rather than technical capabilities. **As components often nest within one another**, each of the components in Figure in the domain partitioning (for example, CatalogCheckout) may use a persistence library and have a separate layer for business rules, **but the top-level partitioning revolves around domains**.

One of the fundamental distinctions between different architecture patterns is what type of top-level partitioning each supports. It also has a huge impact on how an architect decides **— does the architect want to partition things technically or by domain?**

Technical Partitioning

Technical partitioning has one of the organizing principles which is **separation of technical concerns**. This in turn creates useful **levels of decoupling**: if the service layer is only connected to the persistence layer below and business rules layer above, then changes in persistence will only potentially affect those layers. It is certainly logical to organize systems using technical partitioning, but of course this offers some trade-offs.

The **separation enforced by technical partitioning enables developers to find certain categories of the code base quickly**. However, most **realistic software systems require workflows that cut across technical capabilities**. Consider the common business workflow of CatalogCheckout. **The code to handle CatalogCheckout in the technically layered architecture appears in all the layers**, as shown in the following Figure.

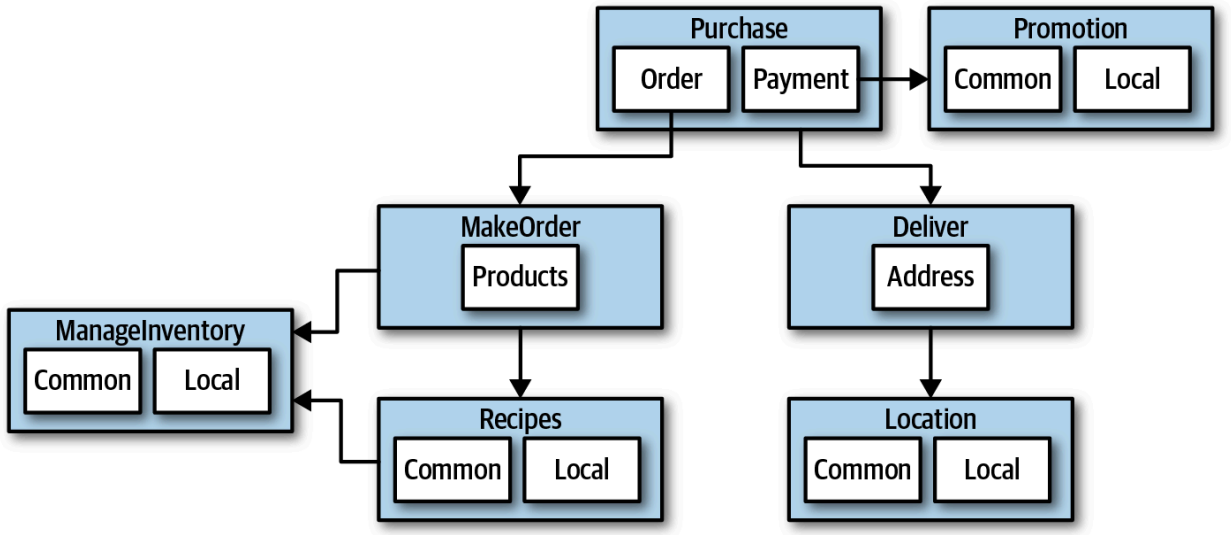


Contrast this with domain partitioning, which uses a top-level partitioning that organizes components by domain.
Each component in the domain partitioning may have subcomponents, including layers, **but the top-level partitioning focuses on domains, which better reflects the kinds of changes that most often occur on projects.**

Neither of these styles is more correct than the other—refer to the First Law of Software Architecture. That said, we have observed a decided industry **trend over the last few years toward domain partitioning** for the monolithic and distributed (for example, **microservices**) architectures. However, it is **one of the first decisions an architect must make**.

Case Study: Silicon Sandwiches: Partitioning

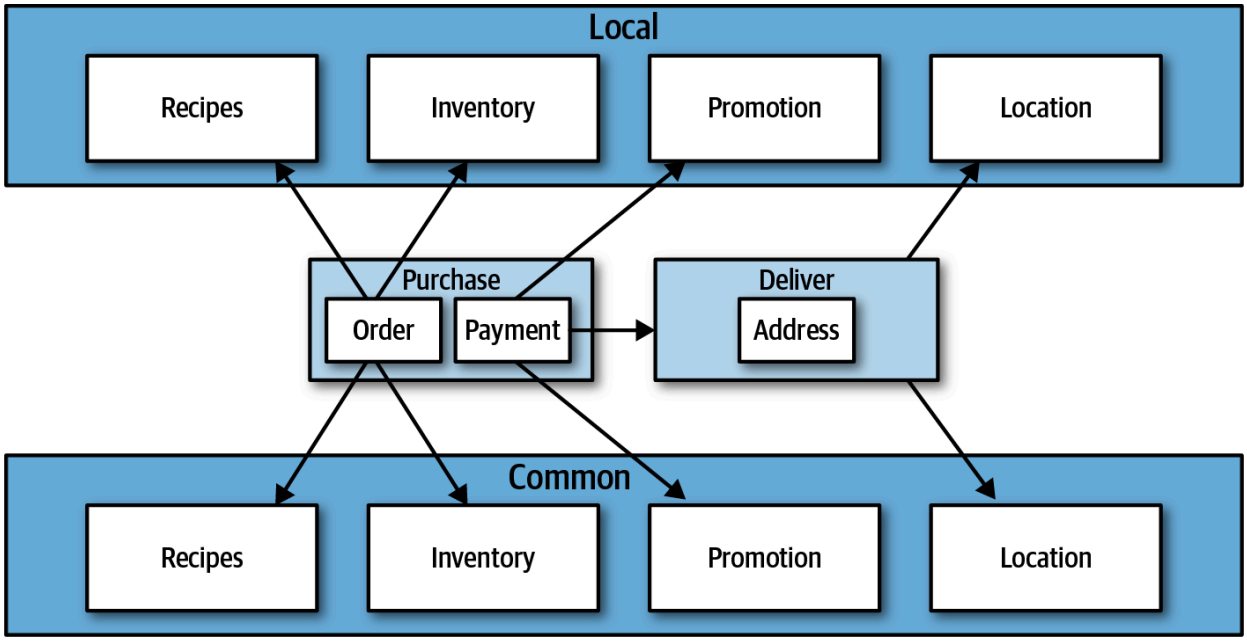
When deriving components, one of the **fundamental decisions facing an architect is the top-level partitioning.** Consider the first of two different possibilities for Silicon Sandwiches, a domain partitioning.



In the previous Figure , the architect has designed **around domains (workflows), creating discrete components** for Purchase, Promotion, MakeOrder, ManageInventory, Recipes, Delivery, and Location. Within many of these components **resides a subcomponent to handle the various types of customization required**, covering both common and local variations.

An alternative design isolates the common and local parts into their own partition, illustrated in the next Figure. Common and Local represent top-level components, with Purchase and Delivery remaining to handle the workflow.

Which is better? It depends! Each partitioning offers different advantages and drawbacks.



Domain partitioning

Domain-partitioned architectures separate **top-level** components by **workflows and/or domain**s.

Advantages

- Modeled more closely toward **how the business functions** and **message flow matches the problem domain.**
- Easier to **build cross-functional teams** around domains (Conways' Law).
- Easy to be aligned and migrate data and components to **distributed architecture (the modular monolith and microservices architecture styles)**.

Disadvantage

- **Customization** code appears in **multiple places**.

Technical partitioning

Technically partitioned architectures separate top-level components based on technical capabilities. This may manifest as layers inspired by Model-View-Controller separation or some other ad hoc technical partitioning. The previous Figure **separates components** based on **customization**.

Advantages

- Clearly separates customization code.
- Aligns more closely to the **layered architecture pattern**.

Disadvantages

- **Higher degree of global coupling as** changes to local or common components **may affect all the others**.
- Developers may have to **duplicate domain concepts** in both common and local layers.
- **Typically higher coupling at the data level**. In a system like this, the architects would likely **create a single database**, including **customization and domains**. That in turn **creates difficulties in untangling the data relationships if the architects later want to migrate this architecture to a distributed system.**

Many other factors contribute to an architect's decision on what architecture style to base their design upon, covered in Architecture Styles Chapter.

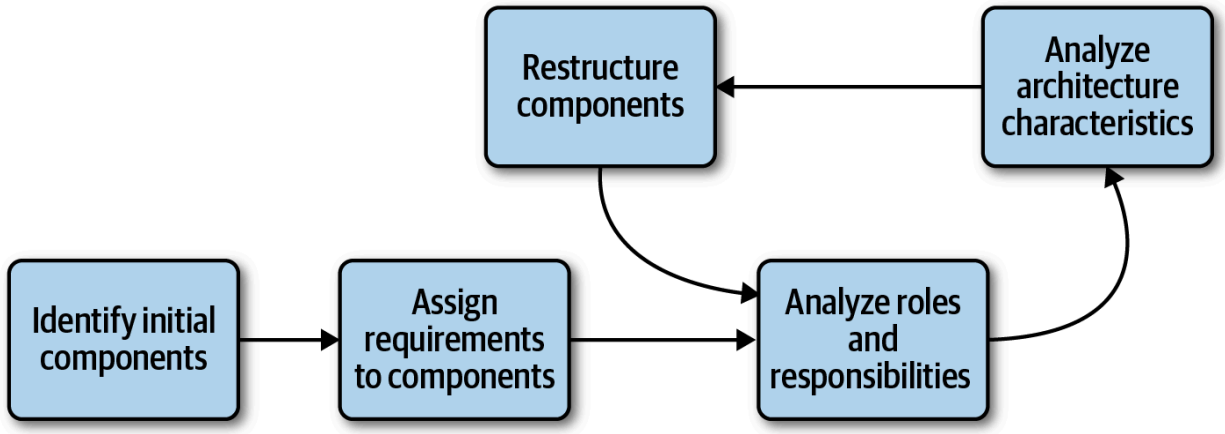
Developer Role

Developers typically take components, designed with the architect role, and further **subdivide them into classes, functions, or subcomponents**. **In general, class and function design is the shared responsibility of architects, tech leads, and mostly developers**.

Developers should never take components designed by architects as the last word; all software design benefits from iteration. **Rather, that initial design should be viewed as a first draft**, where implementation will reveal more details and refinements.

Component Identification Flow

Component identification works best as an **iterative process**, producing candidates and refinements through feedback, illustrated in Figure.



This cycle describes a generic architecture exposition cycle and can be changed or accommodated to certain specialized domains.

Identifying Initial Components

Before any code exists, the architect must determine what top-level components to begin with, based on partitioning they choose. Outside that, an architect should map domain functionality to the components to see where behavior should reside. The likelihood of achieving a good design from this initial set of components is small**, that's why architects must iterate on component design to improve it**.

Align Requirements to Components

The next step aligns requirements (or user stories) to those components to see how well they fit. **This may entail creating new components, consolidating existing ones, or breaking components apart because they have too much responsibility.**

Analyze Components Responsibilities

When assigning stories/requirements to components, the architect should also think about the **roles, responsibilities and behaviors to align the components and domain granularity**. One of the **greatest challenges is discovering the correct granularity**, which **encourages the iterative approach described here**.

Analyze Architecture Characteristics

When assigning stories/requirements to components, **the architect should also look at the architecture characteristics discovered earlier in order to think about how they might impact component division and granularity.** For example, while two parts of a system might deal with user input, the part that deals with hundreds of concurrent users will need different architecture characteristics than another part that needs to support only a few. Thus, while a purely functional view of component design might yield a single component to handle user interaction, **analyzing the architecture characteristics will lead to a subdivision**.

Restructure Components

Feedback is critical in software design. **Thus, architects must continually iterate on their component design with developers**. First, it's virtually **impossible to account for all the different discoveries and edge cases**. Secondly, as the architecture and developers **delve more deeply into building the application, they gain a more nuanced understanding of where behavior and roles should lie.**

Component Design

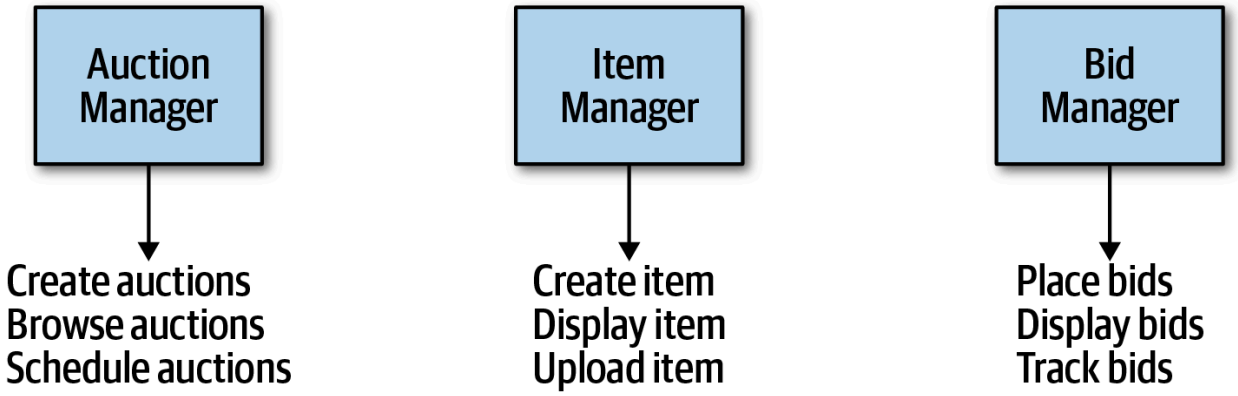
No accepted "correct" way exists to design components. Rather, a wide variety of techniques exist, all with various trade-offs.

Discovering Components

Architects, often in collaboration with other roles such as developers, business analysts, create an **initial component design based on general knowledge** of the system and **based on technical or domain partitioning they chose**. The team goal is to **partition the problem space into coarse chunks that take into account differing architecture characteristics**.

Entity trap

A common anti-pattern may lurk: **the entity trap**. Say that an architect is working on designing components for kata Going, Going, Gone and ends up with a design resembling the next Figure.



The architect has basically taken each entity identified in the requirements and made a Manager component based on that entity. This isn't an architecture; it's an object-relational mapping (ORM) of a framework to a database.
If an architect's needs require merely a simple mapping from a database to a user interface, full-blown architecture isn't necessary; one of the specialized frameworks will suffice.

The entity trap anti-pattern arises when an architect **incorrectly identifies the database relationships as workflows** in the application which indicates **lack of thought about the actual workflows**.

Actor/Actions approach

The actor/actions approach is a popular way that architects use to map requirements to components. In this approach, originally defined by the Rational Unified Process, architects identify **actors/users who perform activities** with the application and **the actions those actors may perform**.
This style of component decomposition works well for all types of systems, monolithic or distributed.

Event storming

Event storming as a component discovery technique comes from domain-driven design (DDD) and shares popularity with microservices. In event storming, the architect assumes **the project will use messages and/or events to communicate between the various components**. The team tries to determine **which events** occur in the system **based on requirements and identified roles, and build components around those event and message handlers**. **This works well in distributed architectures** like microservices that use

events and messages, **because it helps architects define the messages** used in the eventual system.

Workflow approach

An alternative to event storming not using DDD or messaging. The workflow approach is by identifying the key roles and their workflows and modeling the components around these workflows, much like event storming, but **without the explicit constraints of building a message-based system**.

None of these techniques is superior to the others; all offer a different set of trade-offs. If a team uses a waterfall approach, they might prefer the Actor/Actions approach because it is general. When using DDD and corresponding architectures like microservices, event storming matches the software development process exactly.

Case Study: Going, Going, Gone: Discovering Components

The Actor/Actions approach works well as a generic solution. It's the one we use in our case study for Going, Going, Gone.

In Chapter "Scope of Architecture characteristics", we discussed the architecture kata for Going, Going, Gone (GGG) and discovered architecture characteristics for this system. **This system has three obvious roles: the bidder, the auctioneer, and a frequent participant and the system, for internal actions.** For example, in GGG, once the auction is complete, the system triggers the payment system to process payments.

We can also identify a starting set of actions for each of these roles:

Bidder

- View live video stream, view live bid stream, place a bid

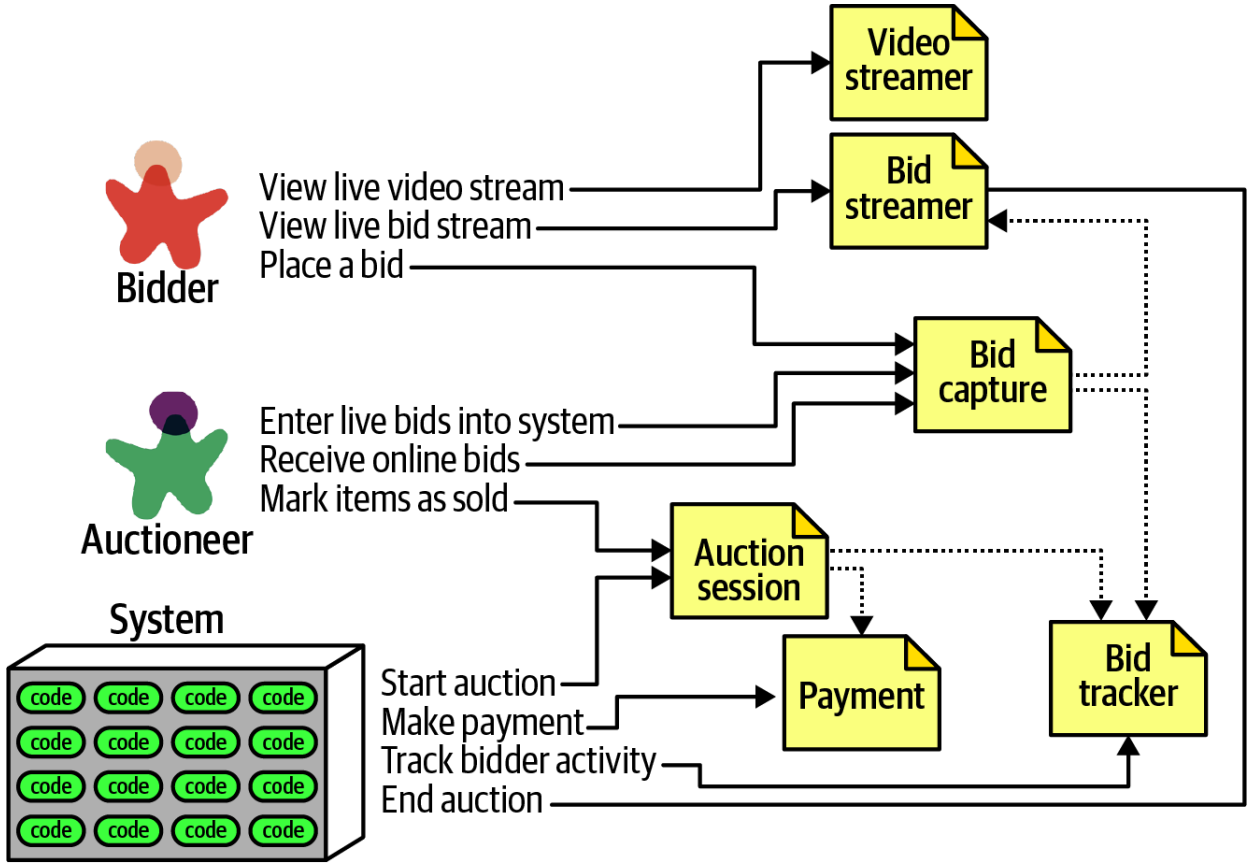
Auctioneer

- Enter live bids into system, receive online bids, mark item as sold

System

- Start auction, make payment, track bidder activity

Given these actions, we can iteratively build a set of starter components for GGG; one such solution appears in the following Figure.



In the Figure, each of the roles and actions maps to a component, which in turn may need to collaborate on information. These are the components identified:

VideoStreamer

- Streams a live auction to users.

BidStreamer

- Streams bids as they occur to the users. Both VideoStreamer and BidStreamer offer read-only views of the auction to the bidder.

BidCapture

- This component captures bids from both the auctioneer and bidders.

BidTracker

- Tracks bids and acts as the system of record.

AuctionSession

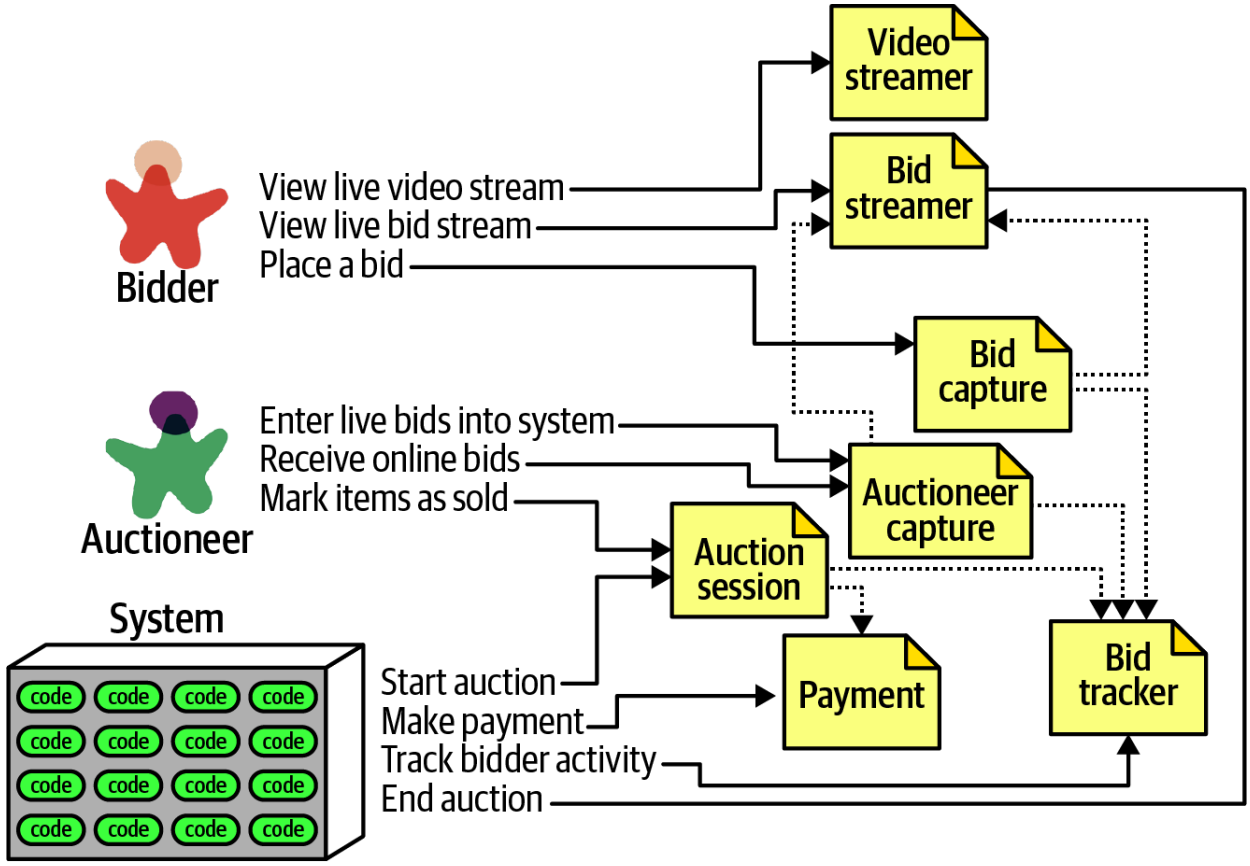
- Starts and stops an auction. When the bidder ends the auction, performs the payment and resolution steps, including notifying bidders of the ending.

Payment

- Third-party payment processor for credit card payments.

Referring to the component identification flow diagram previously discussed, **after the initial identification of components, the architect next analyzes architecture characteristics to determine if that will change the design**. For example, the current design features a BidCapture component to **capture bids from both bidders and the auctioneer,** which makes sense functionally.
However, what about architecture characteristics around bid capture? The auctioneer doesn't need the same level of scalability or elasticity as potentially thousands of bidders. An architect should also **ensure that characteristics like reliability** (connections don't drop) and **availability** (the system is up) **for the auctioneer could be higher** than other parts of the system.

Because they have differing levels of architecture characteristics, the architect decides to **split the Bid Capture component into Bid Capture and Auctioneer Capture**. The updated design appears in the next Figure.



The architect creates a new component for Auctioneer Capture. Note that **Bid Tracker is now the component that will unify the two very different information streams**: the **single** stream of information from the **auctioneer** and the **multiple streams from bidders**.

This design isn't likely the final design but a good starting point to start iterating further on the design.

As an architect, don't obsess over finding the one true design, because many will suffice **(and less likely overengineered)**.
Rather, try to objectively assess the trade-offs between different design decisions, and choose the one that has the least worst set of trade-offs.

Architecture Styles

The **difference** between an **architecture style** and an **architecture pattern** can be confusing. **We define an architecture style as the overarching structure of how the user interface and backend source code are organized and how that source code interacts with a datastore. Architecture design patterns**, on the other hand, are **lower-level** design structures that help form specific **solutions within an architecture style** (**such as how to achieve high scalability or high performance between sets of services**).

Architects must understand the various styles and the trade-offs encapsulated within each to make effective decisions for each particular business problem.

Fundamental Patterns (Not Design Patterns!)

Big Ball of Mud

Architects refer to the **absence of any discernible architectural structure as a Big Ball of Mud** (**anti-pattern)**.
In modern terms, a big ball of mud might describe a simple scripting application with event handlers wired directly to database calls, with no real internal structure. Many trivial applications start like this then become unwieldy as they continue to grow.

In general, **architects want to avoid this type of architecture at all costs.** The lack of structure makes **change increasingly difficult**. This type of architecture also suffers from **problems in deployment, testability, scalability, and performance**.

Unitary Architecture

When software originated, there was only the computer, and software ran on it. For example, when personal computers first appeared, much of the commercial development focused on single machines. As networking PCs became common, distributed systems (such as client/server) appeared.

Client/Server

A fundamental style in architecture separates technical functionality between **frontend and backend**, called a **two-tier, or client/server**, architecture. Many different flavors of this architecture exist, depending on the era and computing capabilities. Such as Desktop + database server, Browser + web server.

Monolithic Versus Distributed Architectures

Architecture **styles can be classified into two main types: monolithic (single deployment unit of all code) and distributed (multiple deployment units connected through remote access protocols).**

Monolithic

- Layered architecture
- Pipeline architecture
- Microkernel architecture

Distributed

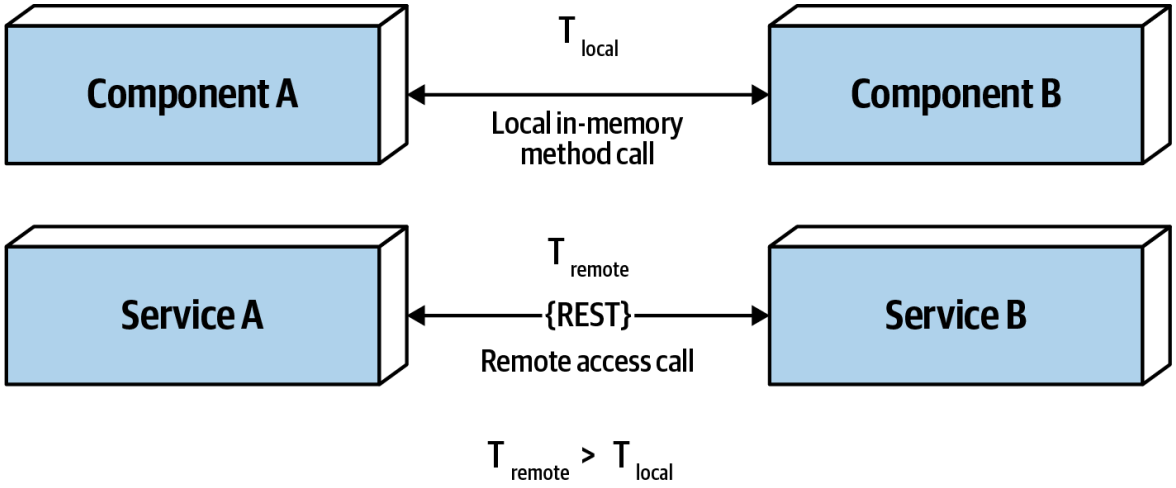
- Service-based architecture
- Event-driven architecture
- Space-based architecture
- Service-oriented architecture
- Microservices architecture

Distributed architecture styles, while being much more powerful in terms of performance, scalability, and availability than monolithic architecture styles, have significant trade-offs for this power. The first group of issues facing all distributed architectures are described in the fallacies of distributed computing.
A fallacy is something that is **believed or assumed to be true but is not**. All eight of the fallacies of distributed computing apply to distributed architectures today. The following sections describe each fallacy.

Fallacy #1: The Network Is Reliable

Developers and architects alike assume that the network is reliable, but it is not. While networks have become more reliable over time, the fact of the matter is that networks **still remain generally unreliable**.
This is significant for all distributed architectures because all distributed architecture styles rely on the **network for communication to, from and between services**.
This is why things like **timeouts and circuit breakers** exist between services.

Fallacy #2: Latency Is Zero



As the Figure shows, when a local call is made to another component via a **method or function call**, that time (**t_local**) is **measured in nanoseconds or microseconds**. However, when that same call is made through a **remote access protocol** (such as REST, messaging, or RPC) **t_remote is measured in milliseconds**.
Therefore, t_remote will always be greater than t_local. **Latency in any distributed architecture is not zero**, yet most architects ignore this fallacy. Ask yourself this question: do you know what the average round-trip latency is for a RESTful call in your production environment? Is it 60 milliseconds? Is it 500 milliseconds?

When using any distributed architecture, **architects must know this latency average. It is the only way of determining whether a distributed architecture is feasible**, particularly microservices because of the amount of communication between those services. For example, assuming an average of 100 milliseconds of latency per request, chaining together 10 service calls adds 1,000 milliseconds to the request!.

Fallacy #3: Bandwidth Is Infinite

Bandwidth is usually not a concern in monolithic architectures as little or no bandwidth is required to process that business request.
But in a **distributed architecture** such as microservices, communication to and between these services **significantly utilizes bandwidth**, impacting latency (fallacy #2) and reliability (fallacy #1).

To illustrate the importance of this fallacy, consider two services. Let's say a **service manages the wish list items** for the website, and **a service manages the customer profile**. Whenever a **request for a wish list** comes, it must make an interservice **call to the customer profile service to get the customer name in the wish list response contract**.
The **customer profile service** returns data **totaling 500 kb** to the wish list service, **which only needs the name (200 bytes)**. **This is a form of coupling referred to as stamp coupling**.
Requests for the wish list items **happen** about **2,000 times a second**. The total amount of **bandwidth** used for that one interservice call is **1 Gb!**
If the customer profile service **were to only pass back the data needed** by the wish list service (**200 bytes**), **the total bandwidth is only 400 kb**.

***Stamp coupling* can be resolved in the following ways:**

- Create private RESTful API endpoints
- Use field selectors in the contract
- Use GraphQL to decouple contracts
- Use value-driven contracts with consumer-driven contracts (CDCs)
- Use internal messaging endpoints

Regardless of the technique used, **ensuring that the minimal amount of data is passed** between services or systems in a distributed architecture is the best way to address this fallacy.

Fallacy #4: The Topology Never Changes

This fallacy refers to the overall network topology, including all of the routers, hubs, switches, firewalls, networks,.. the overall network. **Architects assume that the topology is fixed and never changes. Of course it changes**. What is the significance of this fallacy?

Suppose an architect comes into work on a Monday morning, and everyone is running around like crazy because services keep timing out in production. The architect works with the teams, frantically trying to figure out why this is happening. No new services were deployed over the weekend. What could it be? After several hours the architect discovers that a minor network upgrade happened at 2 a.m. that morning. This supposedly "minor" network upgrade invalidated all of the latency assumptions, triggering timeouts and circuit breakers.

Architects must be in constant communication with operations and network administrators to know what is changing to reduce the type of surprise previously described.

*Architects all the time assume they only need to collaborate and communicate with **one administrator**.*
There are dozens of network administrators in a typical large company. Who should the architect talk to with regard to latency or topology changes.

This fallacy points to the **complexity of distributed architecture and the amount of coordination that must happen to get everything working correctly. Monolithic applications do not require this level of communication and collaboration due to the single deployment unit characteristics of those architecture styles.**

Fallacy #5: Transport Cost Is Zero

Transport cost here does not refer to latency, **but rather to actual cost in terms of money associated with making a "simple RESTful call**". **Distributed architectures cost significantly more than monolithic architectures**, primarily due to **increased needs for additional hardware, servers, gateways, firewalls, new subnets, proxies, and so on.**

Distributed logging

Performing root-cause analysis to determine why a particular order was dropped is **very difficult and time-consuming in a distributed architecture** due to the **distribution of application and system logs**.
Distributed architectures contain dozens to hundreds of different logs, all located in a different place and all with a different format, making it difficult to track down a problem.

Logging consolidation tools such as Splunk **help to consolidate information from various sources and systems** together into one consolidated log and console.

Distributed transactions

Architects and developers **take transactions for granted in a monolithic** architecture world because they are so straightforward and easy to manage. **Standard commits and rollbacks executed from persistence frameworks** leverage ACID (atomicity, consistency, isolation, durability) transactions to **ensure high data consistency and integrity**.
Such is not the case with distributed architectures.

Distributed architectures rely on what is called eventual consistency to ensure the data processed by separate deployment units into a consistent state.
This is one of the trade-offs of distributed architecture: high scalability, performance, and availability at the sacrifice of data consistency and data integrity.

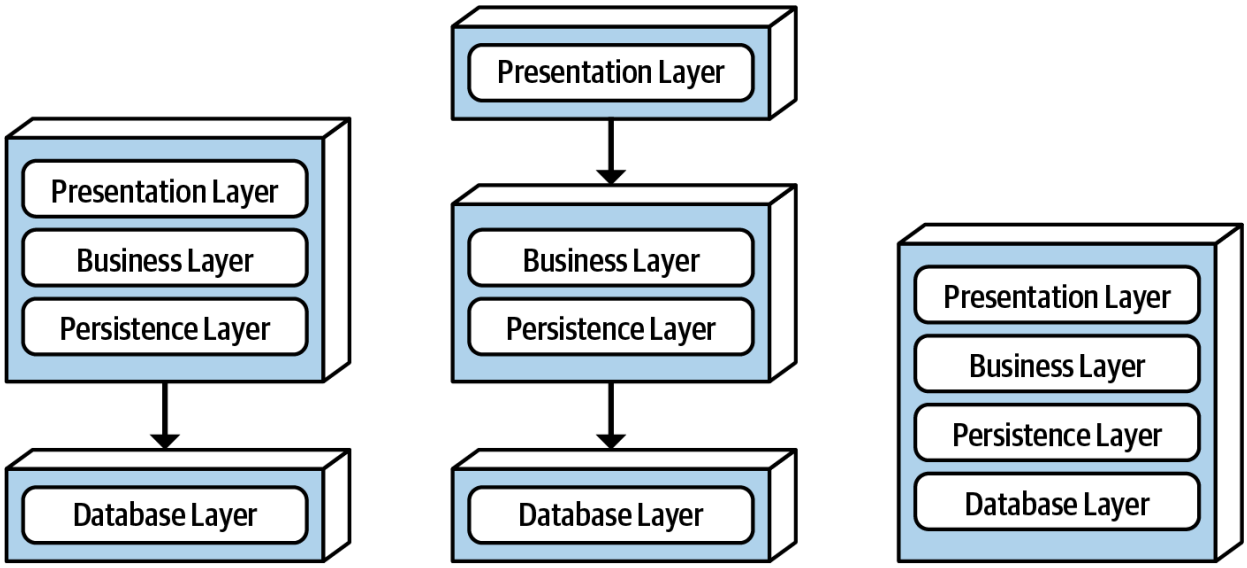
Transactional sagas are one way to manage distributed transactions. Sagas utilize either **event sourcing** to manage the state of transactions.

Layered Architecture Style

The layered architecture, also known as the **n-tiered architecture style**, is one of the most common architecture styles. This style of architecture is the **de facto standard** for most applications, primarily because of **its simplicity, familiarity, and low cost**.
It is also a very natural way to develop applications due to Conway's law as in most organizations there are user interface (UI) developers, backend developers, rules developers, and database experts (DBAs). These organizational layers fit nicely into the tiers of a traditional layered architecture.
If a developer or architect is unsure which architecture style they are using, or if an Agile development team "just starts coding," **chances are good that it is the layered architecture style they are implementing**.

Topology

Components within the layered architecture style are **organized into logical horizontal layers** (technical partitioning), with each layer performing a specific role within the application (such as presentation logic or business logic). Although there are **no specific restrictions in terms of the number and types of layers**, most layered architectures consist of **four standard layers: presentation, business, persistence, and database**, as illustrated in the Figure.



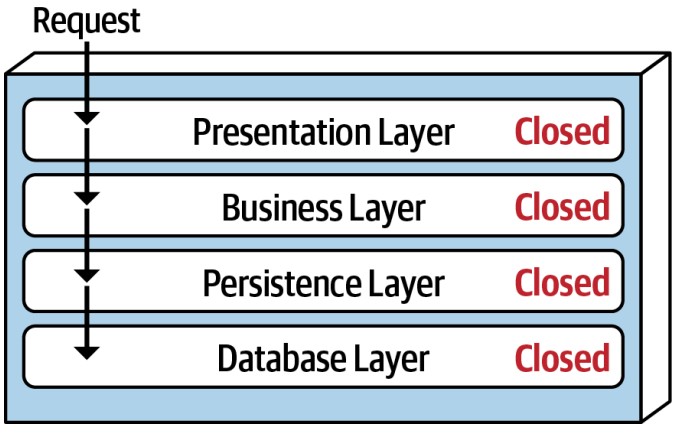
The figure illustrates the **various topology variants** from a physical layering **(deployment) perspective**.
The first variant combines the presentation, business, and persistence layers into a **single deployment unit**, with the database layer typically represented as a **separate external physical database** (or filesystem). The second variant physically separates the **presentation layer into its own deployment unit**, with the business and persistence layers combined into a second deployment unit.The database layer is also usually physically separated through an external database or filesystem. A third variant combines **all four standard layers into a single deployment**, including the database layer. **This variant might be useful for smaller applications**.

Each layer of the layered architecture style has **a specific role and responsibility** forming an abstraction around the work to satisfy a particular business request.
For example, **the presentation layer** would be responsible for **handling all user interface and browser communication logic**, whereas the **business layer** would be responsible for **executing specific business rules associated with the request**. The presentation layer doesn't need to know about how to get customer data. Similarly, the business layer doesn't need to be concerned about how to format customer data for display; it only needs to **get the data from the persistence layer**, **perform business logic** against the data, and **pass that information up to the presentation layer**.

This separation of concerns concept within the layered architecture style makes it easy to build effective roles and **responsibility models** within the architecture. This allows developers to leverage their particular **technical expertise to focus on the technical aspects of the domain** (such as presentation logic or persistence logic). The trade-off of this benefit, however, is a lack of overall agility (the ability to respond quickly to change).

The layered architecture is a technically partitioned architecture (as opposed to a domain-partitioned architecture).
Components are grouped by their technical role in the architecture. Any particular business domain is spread throughout all of the layers of the architecture. For example, the domain of "customer" is contained in the presentation layer, business layer, services layer, and database layer, **making it difficult to apply changes to that domain**. As a result, a **domain-driven design approach does not work as well with the layered architecture style.**

Layers of Isolation



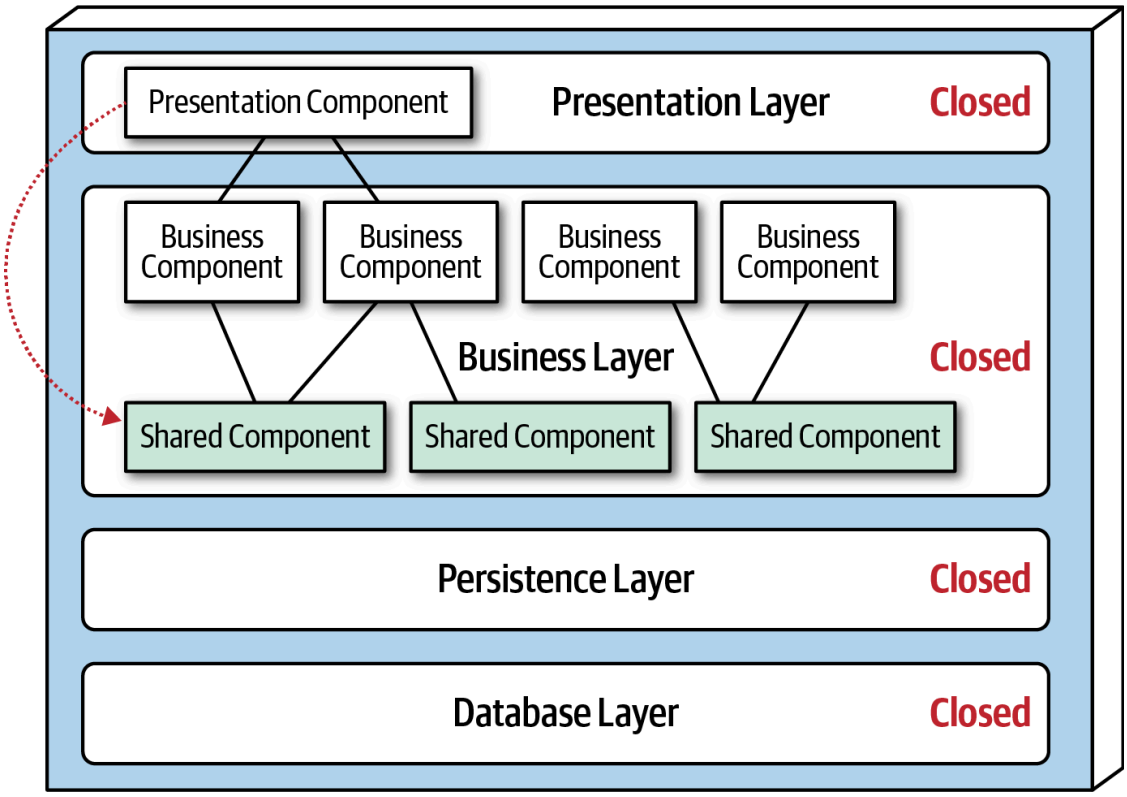
The layers of isolation concept means that **changes made in one layer of the architecture generally don't impact or affect components in other layers, providing the contracts** between those layers remain unchanged.
It also allows any layer in the architecture to be replaced without impacting any other layer.
To support layers of isolation, **layers** involved with the flow of the request necessarily **have to be closed**.
A closed layer means that as a request moves top-down from layer to layer in one direction, the request **cannot skip any layers**, but rather must go through layer by layer.

For example, If the presentation layer can directly access the persistence layer, then changes made to the persistence layer would impact both the business layer and the presentation layer, **producing a very tightly coupled application with layer interdependencies between components.** This type of architecture then **becomes difficult and expensive to change.**

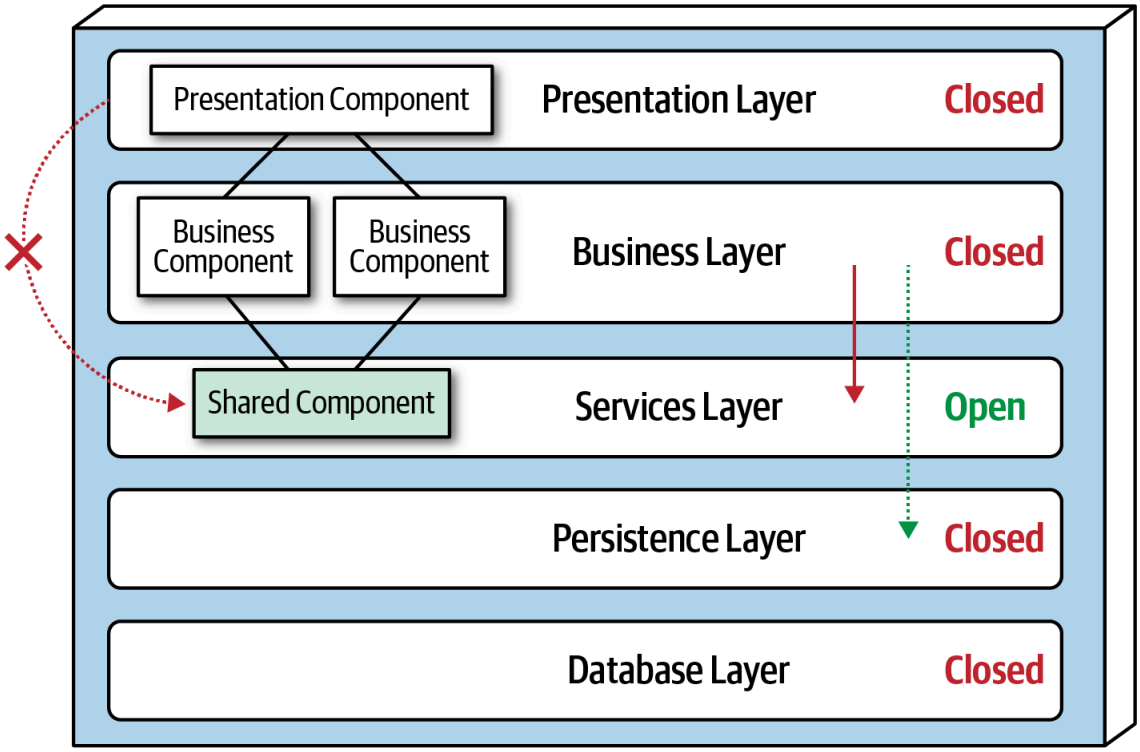
For example, you can leverage the layers of isolation concept within the layered architecture style to replace your older JavaServer Faces (JSF) presentation layer with React.js without impacting any other layer in the application.

Adding Layers

While closed layers facilitate layers of isolation, **there are times when it makes sense for certain layers to be open**. With the dotted line going **from a presentation component to a shared business object** that the request will access such a shared object while **still the request on the presentation layer**. **This scenario is difficult to govern and control because it violates the closed layer** as it should be processed top-down layer by layer (**request processed from presentation to business, not vice versa!**).



One way to architecturally mandate this restriction is to add to the architecture a new services layer containing all of the shared business objects. However, the new services layer must be marked as open; **otherwise the business layer would be forced to go through the services layer to access the persistence layer.**



Failure to document *or properly communicate which* ***layers*** *in the architecture are* ***open and closed*** *(and why) usually results in* ***tightly coupled and brittle architectures*** *that are very difficult to test, maintain, and deploy.*

Why Use This Architecture Style

The layered architecture style is a **good choice for small, simple applications or websites.** Particularly as a starting point, **for situations with very tight budget and time constraints.** Because of the **simplicity and familiarity** among developers and architects, **The layered architecture style is also a good choice when an architect is still analyzing business needs and requirements and is unsure which architecture style would be best.**

Architecture Characteristics Ratings

A one-star rating in the characteristics ratings table (shown in Figure) means the specific architecture characteristic isn't well supported in the architecture.

Architecture characteristic	Star rating
Partitioning type	Technical
Number of quanta	1
Deployability	★
Elasticity	★
Evolutionary	★
Fault tolerance	★
Modularity	★
Overall cost	★★★★★
Performance	★★
Reliability	★★★
Scalability	★
Simplicity	★★★★★
Testability	★★

Overall cost and simplicity are the primary strengths of the layered architecture style. Being monolithic in nature, layered architectures are simple and easy to understand, and are relatively low cost to build and maintain.
However, as a cautionary note, these ratings start to quickly diminish as monolithic layered architectures get bigger and consequently more complex.

Both deployability and testability rates are very low for this architecture style. Deployability rates are low due to the **effort to deploy, high risk**, and **lack of frequent deployments**.
A simple three-line code change to a class file requires the entire deployment unit to be redeployed, taking in potential database changes, configuration changes.
The low testability rating also reflects this scenario; with a simple three-line change, most **developers are not going to spend hours executing the entire regression test suite**, particularly along with dozens of other changes being made at the same time.

We gave testability a two-star rating due to the ability to **mock or stub components** (or even an entire layer), **which eases the overall testing effort**.

Overall reliability rates are medium (three stars), **mostly due to the lack of network traffic**, **bandwidth, and latency found in most distributed architectures and because of the nature of the monolithic deployment in the layered architecture**.

Elasticity and scalability rate very low (one star) for the layered architecture, primarily due to **monolithic deployments and the lack of architectural modularity.** Although it is **possible to scale this monolith** more than others, this effort usually requires **very complex design techniques such as multithreading, internal messaging, and other parallel processing techniques this architecture isn't well suited for**. However, because the layered architecture is always a single system quantum, applications can only scale to a certain point.

Performance is always an interesting characteristic and we gave it only **two stars** because of **the lack of parallel processing, closed layering, and the sinkhole architecture anti-pattern.** Like scalability, performance can be addressed **through caching, multithreading,** and the like, but it is **not a natural characteristic** of this architecture style.

Layered architectures don't support fault tolerance due to monolithic deployments and the lack of architectural modularity. If one small part of a layered architecture causes an out-of-memory condition to occur, the entire application unit is impacted and crashes. Furthermore, **overall availability is impacted** due to the **high mean-time-to-recovery** (MTTR) usually experienced by most monolithic applications.

Microservice Architecture

Microservices is an extremely popular architecture style that has gained significant momentum in recent years. In this chapter, we provide an **overview of the important characteristics that set this architecture apart**, both topologically and philosophically.

Background

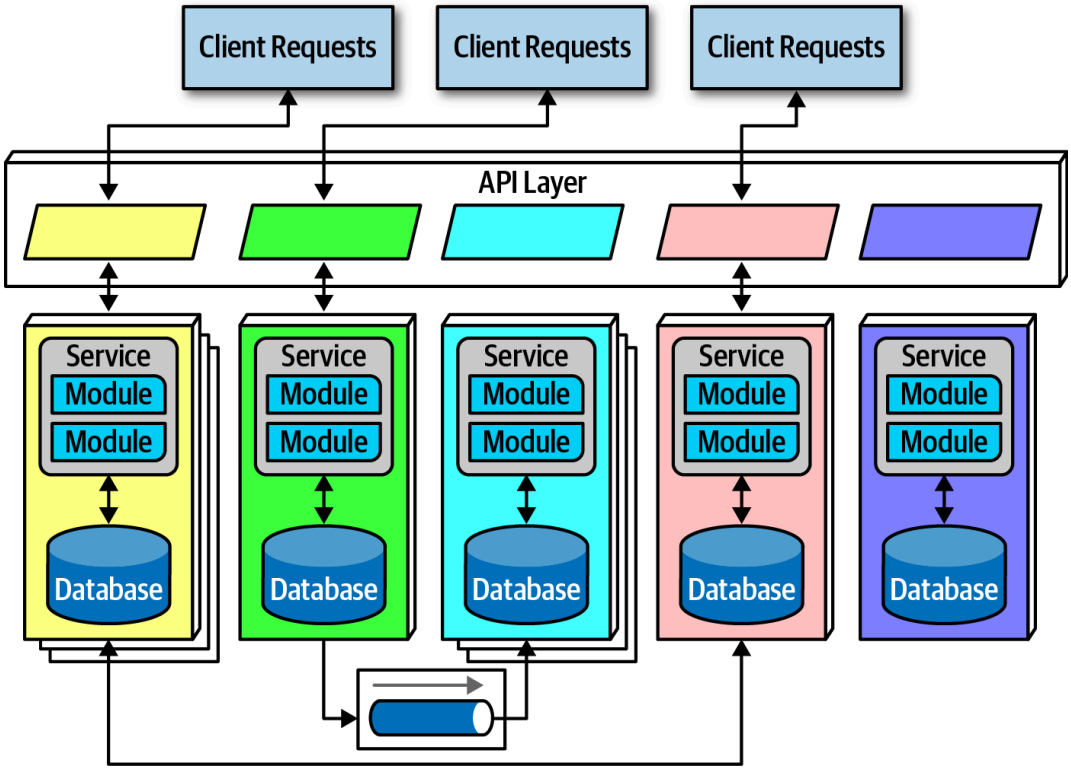
Microservices is heavily inspired by the ideas in domain-driven design (DDD), a logical design process for software projects. **The concept of bounded context of DDD represents a decoupling style**. For example, an application might have a domain called CatalogCheckout, which includes notions such as catalog items, customers, and payment. In a traditional **monolithic architecture, developers would build reusable classes and linked databases**. **Within a bounded context**, **the internal parts**, such as code and data schemas, **are coupled together to produce work; but they are never coupled to anything outside the bounded context**.

While reuse is beneficial, **the negative trade-off of reuse is coupling**.

However, if the architect's goal requires **high degrees of decoupling, then they favor duplication over reuse**.
The primary goal of microservices is high decoupling, physically modeling the logical notion of bounded context.

Topology

The topology of microservices is shown in the following figure.



Due to **its single-purpose nature**, **the service size in microservices is much smaller than other distributed architectures**. Architects expect **each service to include all necessary parts to operate independently**, including databases.

Distributed

Microservices form a distributed architecture**: each service runs in its own process as virtual machines and containers**. Decoupling the services to this degree allows for **heavily feature multitenant infrastructure** for hosting applications. For example, when using a **server to**

manage multiple running applications, it allows **operational reuse of network bandwidth, memory, disk space, and a host of other benefits**. However, if all the applications **continue to grow**, eventually **some resource becomes constrained on the shared infrastructure.** Another problem concerns **improper isolation** between shared applications.

Separating each service into its own process solves all the problems brought on by sharing. **With cloud resources and container technology**, teams can reap the benefits of **extreme decoupling, both at the domain and operational level**.

Performance is often the negative side effect of the distributed nature of microservices. **Network calls** take much longer than method calls, and **security verification at every endpoint** adds additional processing time, *requiring architects to think carefully when designing the system.*

Bounded Context

The driving philosophy of microservices is the notion of bounded context: **each service models a domain or workflow**. **Thus, each service includes everything necessary to operate within the application**, including classes, other subcomponents, and database schemas.

Microservices take the concept of a **domain-partitioned architecture to the extreme**. Microservices is the **physical embodiment of the logical concepts in domain-driven design**.

Granularity (Service Boundary)

Architects struggle to find the correct granularity for services in microservices, and often make the **mistake of making their services too small (too fine-grained)**, which requires them to **build additional communication links** back between the services **to do useful work**.

The purpose of service boundaries in microservices is to capture a domain or workflow.
In some applications, those boundaries **might be large** for some parts of the system.

Here are some guidelines architects can use to help find the appropriate boundaries:

- Purpose

  The most obvious boundary, **a domain**. Ideally, each microservice should be **extremely functionally cohesive**, contributing **one significant behavior** of the overall application.

- Transactions

Bounded contexts are business workflows, and often the entities that need to cooperate in a **transaction show architects a good service boundary**.

- Choreography and Consolidation

  If an architect builds a set of services that offer excellent domain isolation **yet require extensive communication to function**, the architect may **consider bundling these services back into a larger service** *to avoid the communication overhead*.

Remember, Iteration is the only way to ensure good service design. Architects rarely discover the perfect granularity, data dependencies, and communication styles on their first pass.

Data Isolation

Another **requirement of microservices,** driven by the bounded context concept, **is data isolation**. Many other architecture styles use a single database for persistence. However, microservices tries to **avoid all kinds of coupling, including shared schemas and databases**.

Architects must be wary of the *entity trap* and **not** simply model their services to resemble single entities in a database.

Architects **can't create a single source of truth** (as used to do in monolith), **which is no longer an option when distributing data across the architecture**. Thus, architects must decide how they want to handle this problem: **each service can choose the most appropriate tool**, based on price, type of storage, or a host of other factors. Architects can choose a more suitable database (or other dependency) **without affecting other teams**.

API Layer

Most pictures of microservices include an **API layer** sitting between the consumers of the system, **but it is optional**. It is common because **it offers a good location** within the architecture to perform useful tasks, either **via indirection as a proxy or a tie into operational facilities, such as a naming service**.

An API layer **should not be used as a mediator or orchestration tool** to stay true to the underlying philosophy of this architecture: **all interesting logic in this architecture should occur inside a bounded context**, and putting orchestration or other logic in a mediator violates

that rule.
***Architects typically use mediators in technically partitioned architectures*, whereas microservices is firmly domain partitioned.**

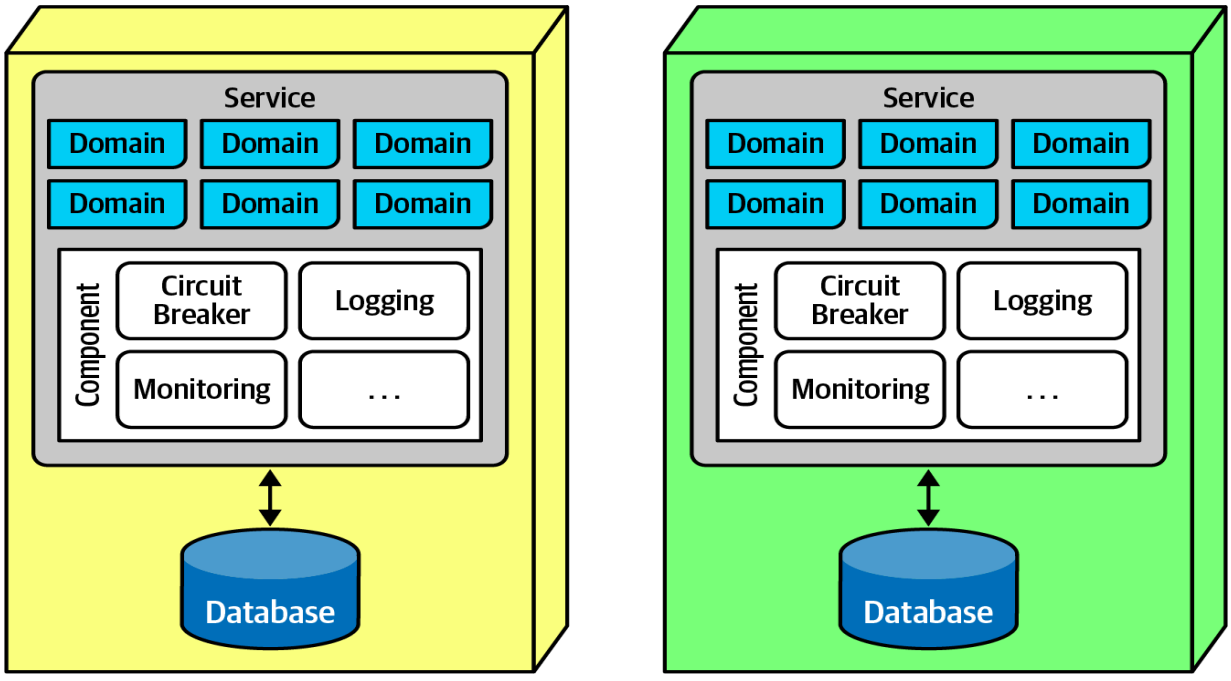
Operational Reuse

SideCar Pattern

Given that **microservices prefer duplication to coupling**, *how do architects **handle** the **parts** that really do **benefit from coupling***, such as **operational concerns** like monitoring, logging, and circuit breakers? In microservices, architects try to split these two concerns.

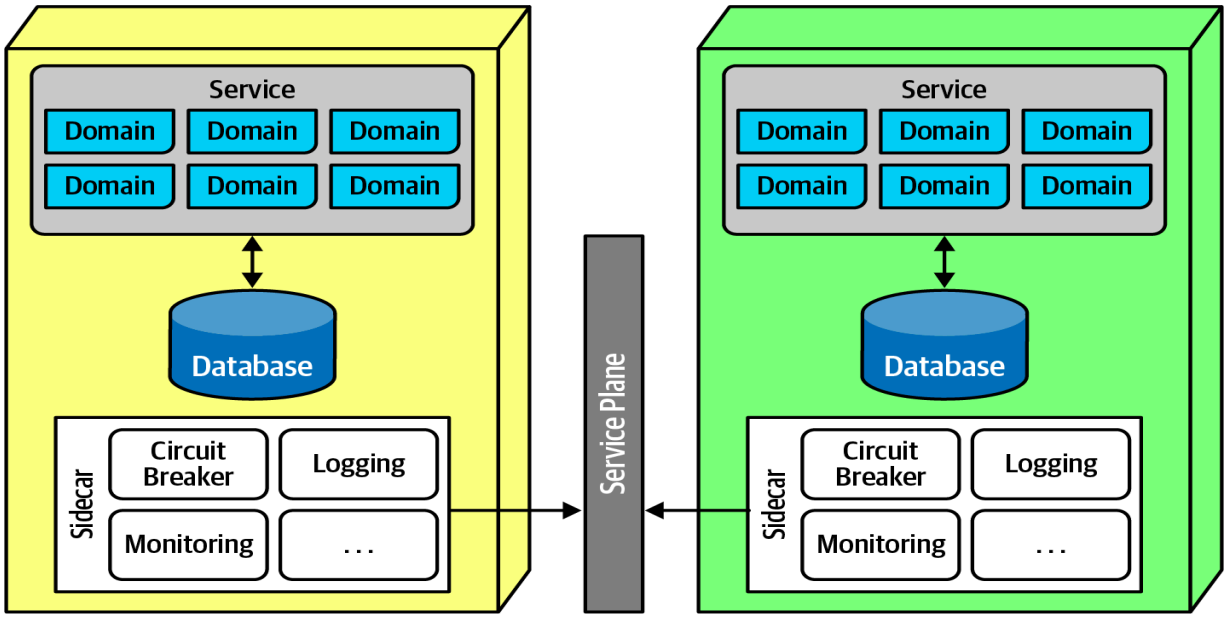
Once microservices are created, teams realize that each has **common elements** that benefit from **similarity**. For example, if an organization allows each service team to implement ***monitoring*** themselves, ***how can they ensure that each team does so and handle monitoring tool upgrades too?***

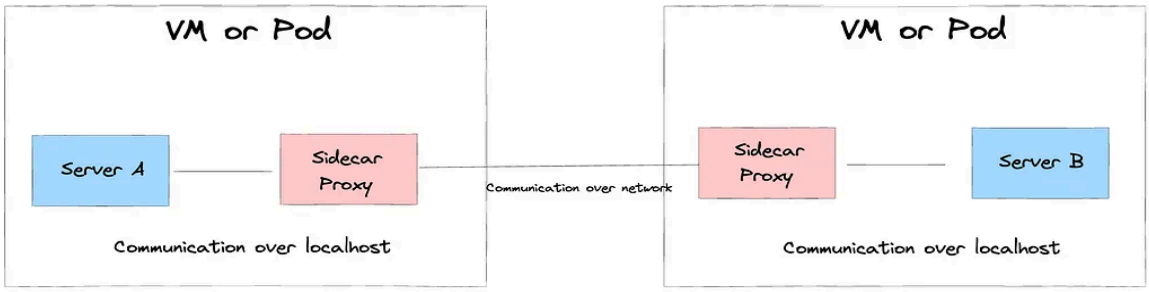
The sidecar pattern offers a solution to this problem.



In the previous figure, **the common operational concerns appear within each service as a component**. **The sidecar component handles all the operational concerns that teams**

benefit from coupling together. Thus, when it comes time to upgrade the monitoring tool, the shared infrastructure team can update the sidecar. As shown in the following figure.





Once teams know that **each service includes a common sidecar** wire into the service plane, they can build a **service mesh**, allowing unified control across the architecture and **all microservices**.

The service mesh forms **a console** that allows teams to **globally control operational coupling,** such as **monitoring levels, logging, and other cross-cutting operational concerns**.

Architects use **service discovery** as a way to build ***elasticity*** into microservices architectures. **A request goes through a service discovery tool, which can monitor the number and frequency of requests**, as well as **spin up new instances of services to handle scale or elasticity concerns**. **Architects often include service discovery in every microservice**. **The API layer is often used to host service discovery, allowing a single place** for user interfaces to find and create services in a consistent way.

How service mesh is different from load balancer

Service meshes provide more advanced and fine-grained control over microservices in distributed architectures. While l**oad balancers focus on distributing traffic**, **service meshes** provide **additional features** such as **service discovery**, security, observability, traffic management, and failure recovery.

What about API Gateway

An organization may choose an **API gateway**, which handles **protocol transactions**, ***instead of a service mesh.*** However, **developers must update the API gateway every time a microservice is added or removed** because API gateway needs to be aware of the available services and their endpoints. Service mesh typically **offers network management scalability and flexibility that exceeds the capabilities of traditional API gateways.**

A service mesh applying the sidecar pattern as a proxy instance, typically in containers and/or microservices. A sidecar is attached to each service. In a container, the sidecar attaches to each application container, VM or container orchestration unit, such as a Kubernetes pod.

Service Mesh Implementation

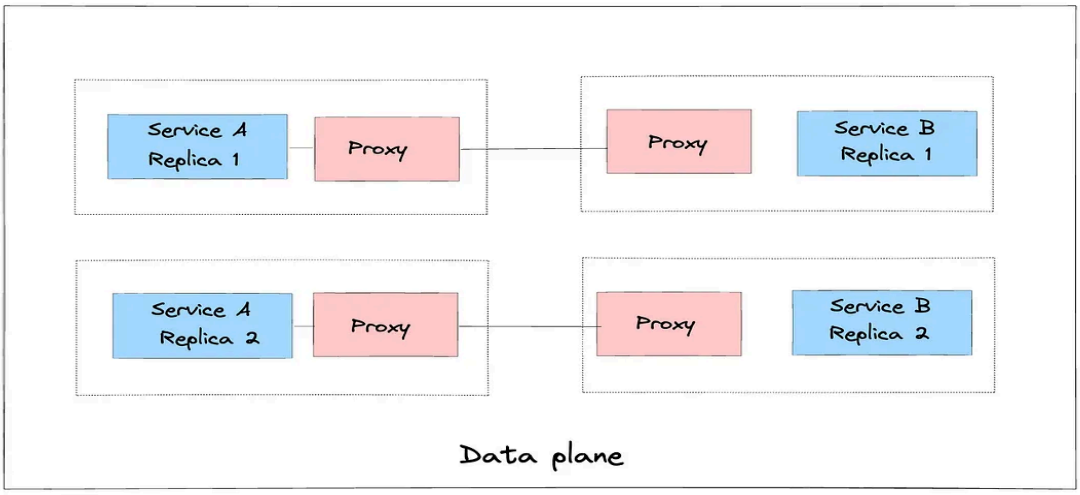
Service mesh implementations usually have two main components:

- Data plane
- Control plane

The Data plane

The data plane is a *network proxy* replicated alongside each microservice (known as a **sidecar**), which manages all inbound and outbound **network traffic** of the microservice. ***As part of this, it may perform service discovery, load balancing, security and reliability functions.***

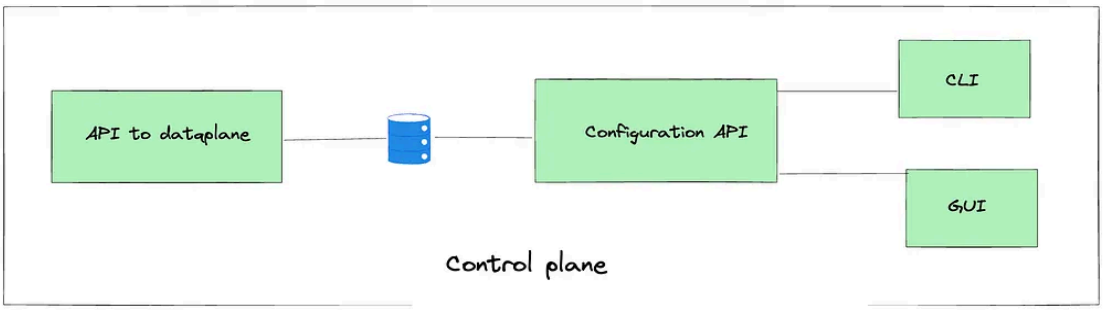
Service instances, sidecars and their interactions make up what is called the **data plane** in a service mesh.



Sidecars are **Layer 7 proxies**. By operating at this layer, service meshes provide capabilities like ***intelligent load balancing based on observed latency***, or provide sensible and consistent timeouts for requests.

The Control Plane

Proxies need to be configured through the control plane, which consists of **several services** that provide **administrative** functionality with interface (**CLI or API**) **to configure the behavior of the data plane** and for the proxies to coordinate their actions.

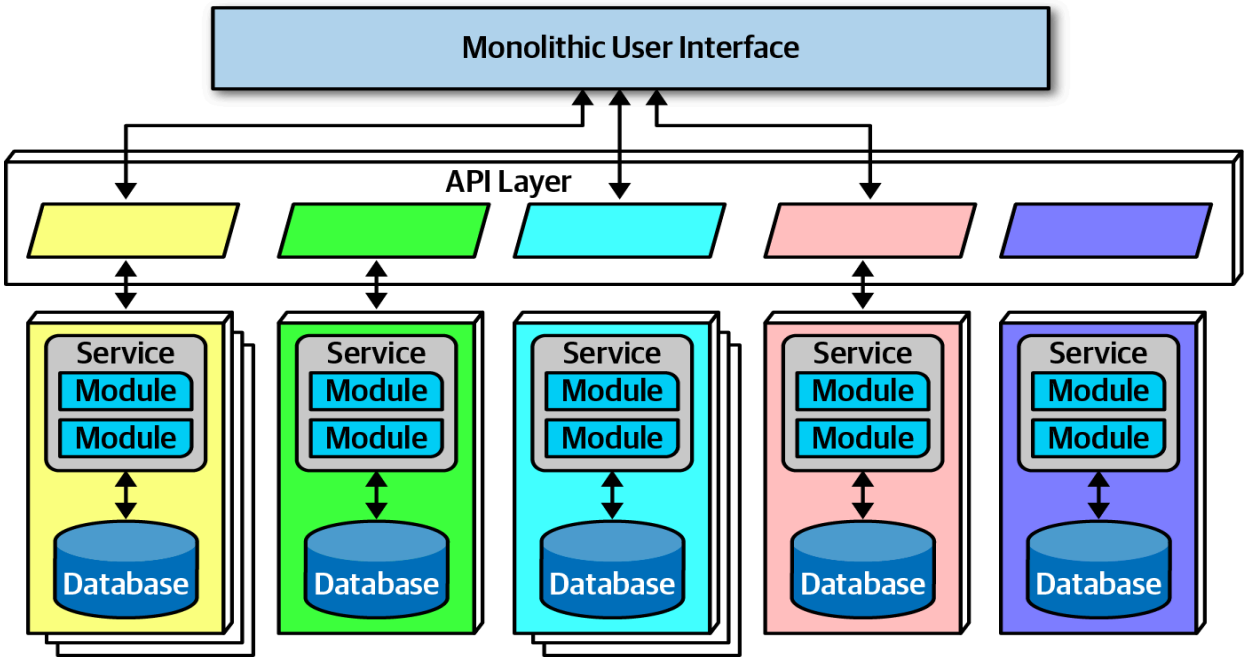


For example, operators work through the **control plane** to define **routing rules, create circuit breakers, or enforce access control.**

Teams can use the control plane to ***export observability data such as metrics, logs, and traces.***

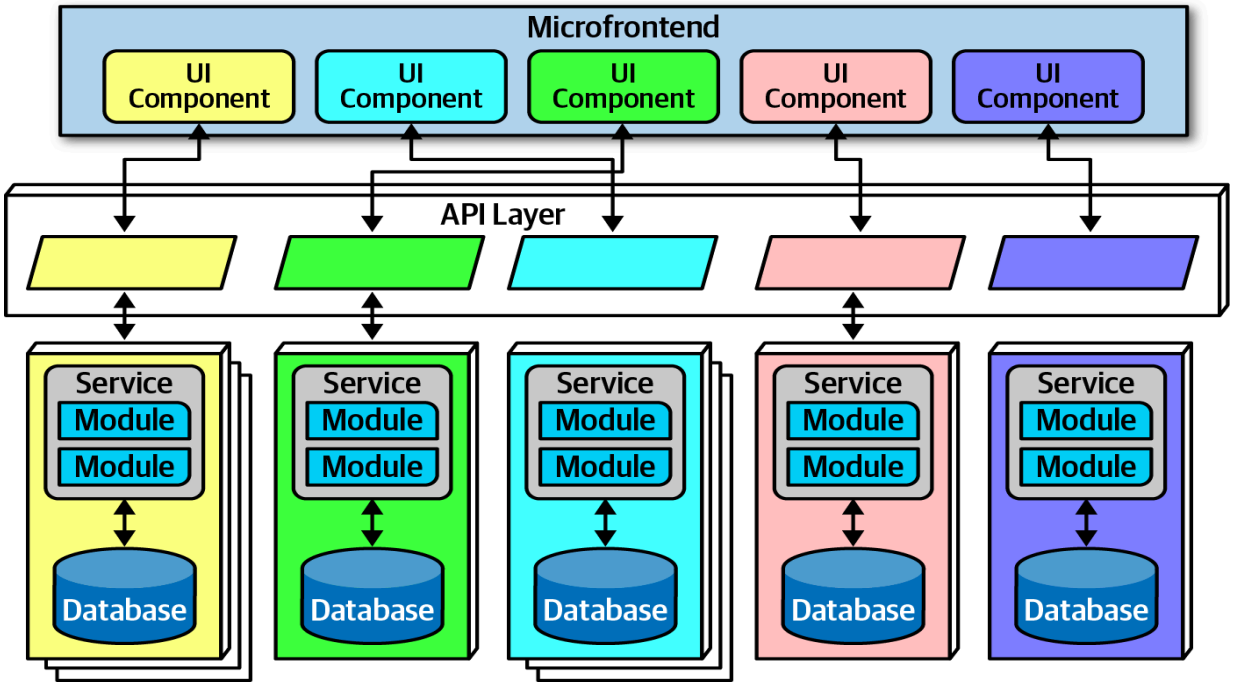
Frontends

Two styles of user interfaces commonly appear for microservices architectures; first is monolith user interface appears in the following figure.



The monolithic frontend calls through the API layer to satisfy user requests. The frontend could be a rich desktop, mobile, or web application.

The second option for user interfaces uses microfrontends as appears in the following figure.



Using this pattern, teams can isolate service boundaries from the user interface to the backend services, unifying the entire domain within a single team.

Communication

In microservices, architects and developers **struggle with appropriate granularity of each service**, which affects both data isolation and communication. **Finding the correct communication style helps teams keep services decoupled** yet still coordinated in useful ways.

Fundamentally, architects must decide on synchronous or asynchronous communication.
Microservices architectures typically utilize **protocol-aware heterogeneous interoperability**.

Protocol-aware

Services must know (or discover) which protocol to use to call other services.

Heterogeneous

Because microservices is a distributed architecture, each service **may be written in a different technology stack**. **Heterogeneous** suggests that microservices fully support different environments.

Interoperability

Describes services calling one another. While architects in microservices try to **discourage *transactional* method calls**, services commonly call other services via the network to collaborate and send/receive information.

ENFORCED HETEROGENEITY

A well-known architect who was a pioneer in the microservices style. Because they had a fast-moving problem domain, the **architect wanted to ensure that none of the development teams accidentally created coupling points** between each other. **He mandated each development team to use a different technology stack.** If one team was using Java and the other was using .NET, ***it was impossible to accidentally share classes!***

This approach is the polar opposite of most enterprise governance policies, which insist on standardizing on a single technology stack. **The goal in the microservices world is to choose the correct scale technology for the narrow scope of the problem.** This concept leverages the ***highly decoupled nature of microservices***.

For asynchronous communication, architects often use events and messages, thus internally utilizing an **event-driven architecture**.

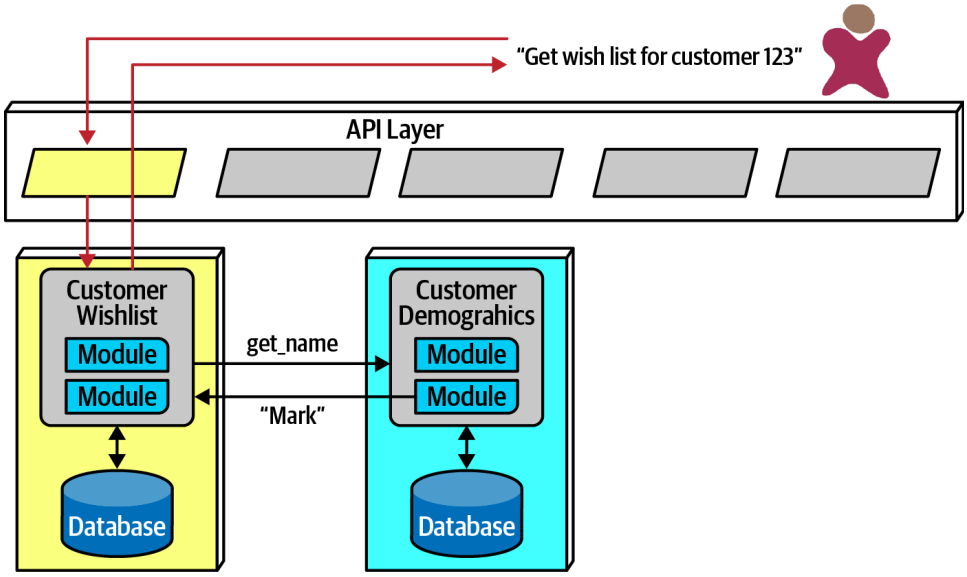
The broker and mediator patterns manifest in **microservices** as ***choreography and orchestration***.

Choreography

Choreography utilizes the same communication style as a broker event-driven architecture. In other words, **no central coordinator exists in this architecture (Decentralized Transaction Management)**.

Thus, architects find it natural to implement ***decoupled events between services (Choreography)***.

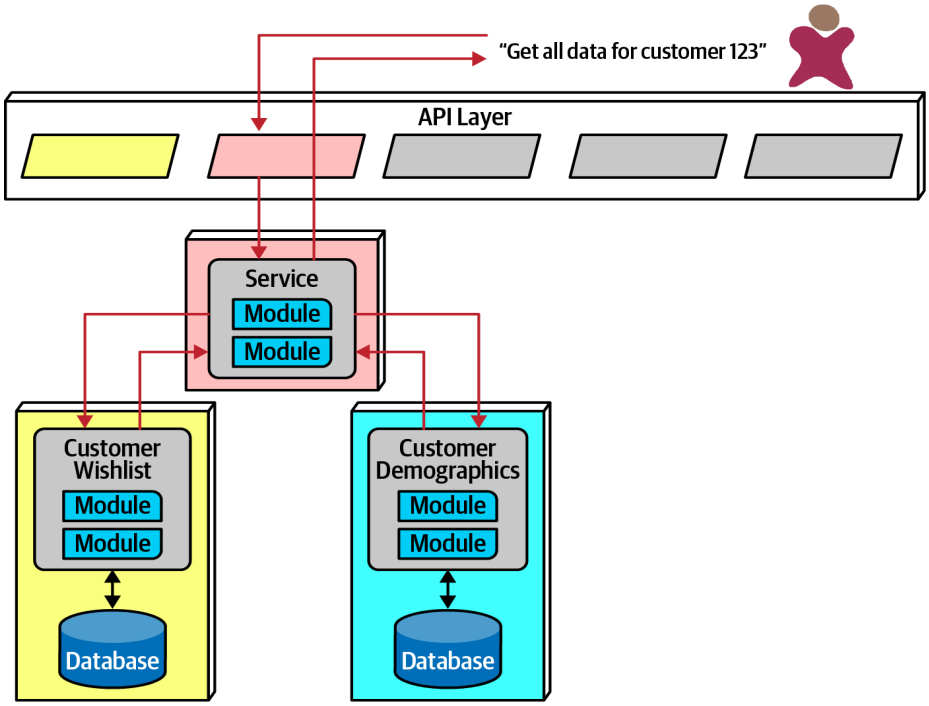
Consider the scenario shown in the following figure.



The user requests details about a user's wish list. Because the CustomerWishList service doesn't contain all the necessary information, it makes a call to CustomerDemographics to retrieve the missing information, returning the result to the user.

Because microservices architectures don't include a global mediator like other service-oriented architectures,

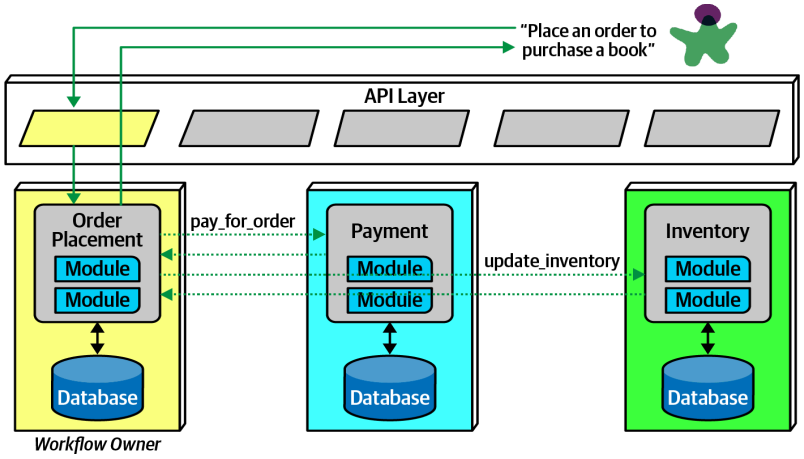
If an architect needs to coordinate across several services,
they can ***create their own localized mediator service*****,** as shown in the following figure.



Orchestration

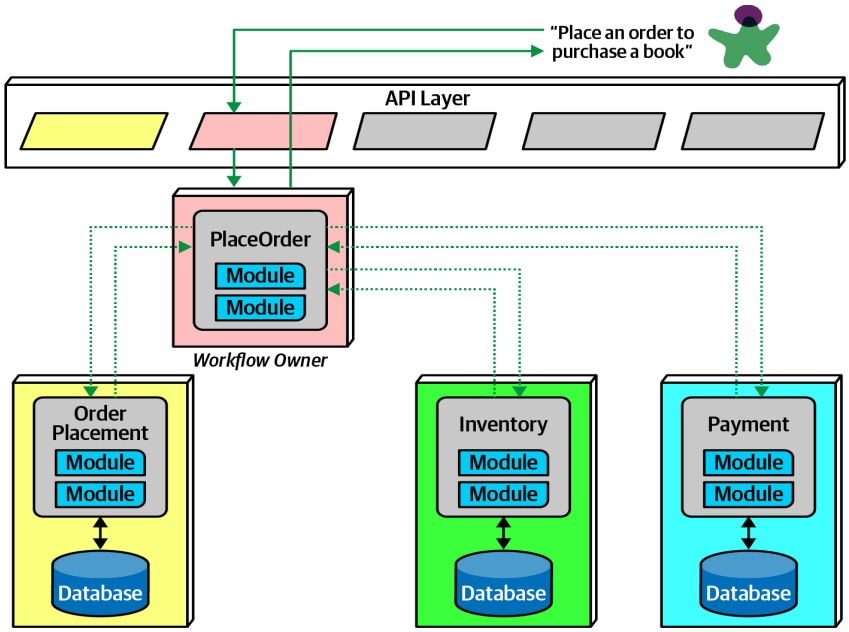
However, common problems like error handling and coordination **become more complex in choreographed environments**.

Consider an example with a more complex workflow, shown in the figure.



The first service called must coordinate across a wide variety of other services, basically acting as a **mediator** in addition to its other domain responsibilities. This pattern is called the front controller pattern.

So alternatively, an architect may choose to use ***orchestration service for complex business processes***, illustrated in the figure.



The architect builds a **mediator (Centralized Transaction Management)** to handle the complexity and coordination required for the business workflow. ***While this creates coupling between these services***, it allows the architect **to focus coordination into a single service**.

Transactions and Sagas

Architects often **encounter the problem of how to do transactional coordination across services**. ***Atomicity*** that was trivial in monolithic applications **becomes a problem in distributed ones.**

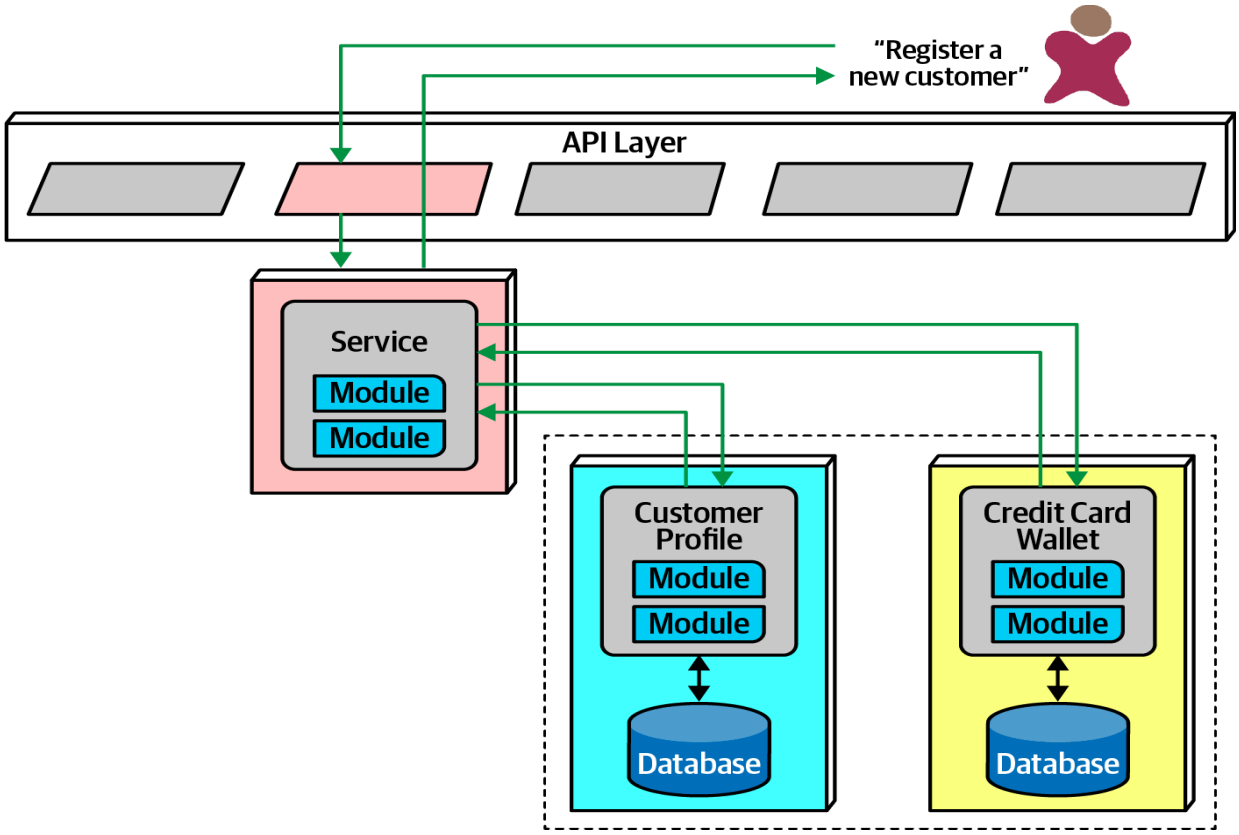
Building transactions across service boundaries violates the core decoupling principle of the microservices architecture (and also creates the ***worst kind of dynamic connascence***, connascence of value).

The best advice *for architects who want to* ***do transactions across services*** *is:* ***don't! – Fix the granularity components instead.***

Exceptions always exist. For example, a situation may arise where two **completely different services** in architecture characteristics and distinct service boundaries, yet **still need**

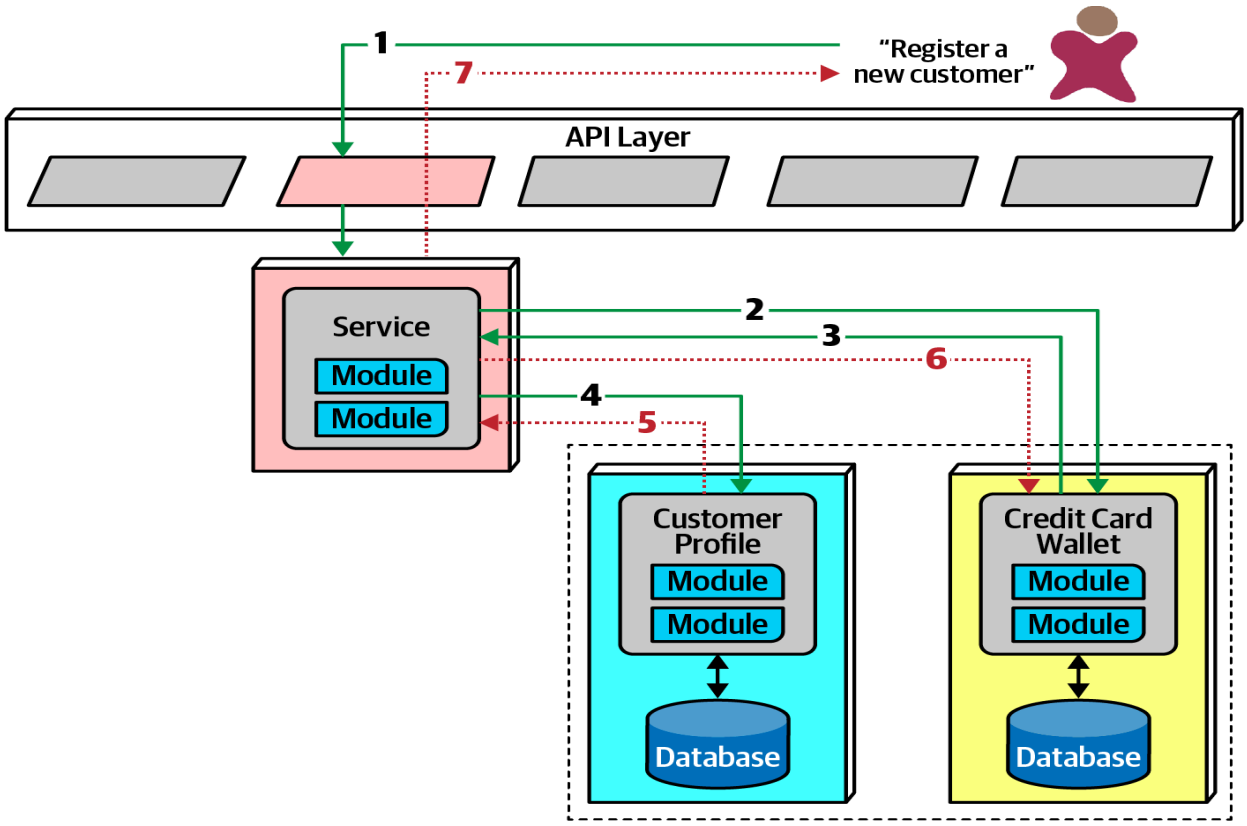
transactional coordination. In those situations, **patterns exist to handle transaction orchestration*, with serious trade-offs.***

A popular distributed transactional pattern in microservices is **the saga pattern**, illustrated in the figure.



A service acts as a **mediator (Centralized or Decentralized Transaction Management)** across multiple service calls and **coordinates the transaction**. The mediator calls each part of the transaction, **records success or failure, and coordinates results**.
If everything goes as planned, all the values in the services and their contained databases **update synchronously.**

In an error condition, **the mediator must ensure that no part of the transaction succeeds if one part fails**.
Consider the situation shown in the figure.



If the first part of the transaction succeeds, **yet the second part fails**, **the mediator must send a request to all the parts of the transaction that were successful and tell them to undo the previous request.** This style is called a ***compensating transaction framework.*** Developers implement this pattern by **usually having each request from the mediator enter a pending state** until the mediator indicates overall success.

*However, this design becomes **complex if many asynchronous requests must exist** and also creates **a lot of coordination traffic at the network level.***

The best advice for architects is to use the saga pattern sparingly.

Summary The Difference

While the Saga pattern, orchestration, and choreography **all involve coordination and communication between services in a distributed system**, they each have **distinct characteristics** and are suited to **different scenarios**. Here's a practical comparison highlighting their differences:

Saga Pattern:

- Use Case: The Saga pattern is used for managing distributed transactions across multiple services, especially in long-running business processes where maintaining data consistency is crucial.
- Approach: **Sagas break down a large transaction into a series of smaller, localized transactions within each service.** These local transactions are coordinated through a series of **compensating actions** to ensure eventual consistency.
- Coordination: **Coordination** in the Saga pattern can be **centralized** (using a Saga orchestrator) **or decentralized** (choreographed), depending on the implementation.
- Example: In an e-commerce platform, when a customer places an order, the Saga pattern may be used to manage transactions across services involved in inventory management, order processing, payment processing, and shipping.

Orchestration:

- Use Case: Orchestration is used for **managing complex business processes** where a **central** controller (**orchestrator**) **determines the flow of operations** and coordinates interactions between services.
- Approach: The orchestrator defines the **sequence of tasks** or steps in the business process and coordinates the execution of these tasks by invoking appropriate services.
- **Centralized** Control: Orchestration provides **centralized control** over the execution flow, making it easier to manage and monitor complex workflows.
- Example: In a travel booking system, an orchestration service might coordinate interactions between services responsible for flight booking, hotel reservation, car rental, and payment processing to fulfill a customer's booking request.

Choreography:

- Use Case: Choreography is used for managing **simpler business processes** where services interact autonomously by **publishing and subscribing to events or messages**.
- Approach: Services **communicate directly with each other through events or messages,** and each service reacts to events emitted by other services to trigger its own actions.
- Decentralized Control: **Choreography decentralizes control, allowing services to operate independently based on events received from other services.**
- Example: In a notification system, when a user subscribes to a topic, a notification service might publish events to a **message broker**. Other services interested in the topic can react to these events, such as sending emails, SMS messages, or push notifications to the user.

Architecture Characteristics Ratings

As microservices is a distributed architecture, **it suffers from many of the deficiencies** inherent in architectures made from **pieces wired together at runtime**. Thus, **fault tolerance and reliability are impacted when too much interservice communication is used. Developers fix many of these problems by redundancy and scaling via service discovery.**

Microservices couldn't exist without the DevOps revolution.
It has the high support for modern engineering practices such as automated deployment, testability, and others (Note: they are not listed).

Ratings for microservices

Architecture characteristic	Star rating
Partitioning type	Domain
Number of quanta	1 to many
Deployability	★★★★
Elasticity	★★★★★
Evolutionary	★★★★★
Fault tolerance	★★★★
Modularity	★★★★★
Overall cost	★
Performance	★★
Reliability	★★★★
Scalability	★★★★★
Simplicity	★
Testability	★★★★

The high points of this architecture are ***scalability, elasticity, and evolutionary***. Because the architecture ***relies heavily on automation and intelligent integration with operations***, developers can also **build elasticity support** into the architecture. Because the architecture favors ***high decoupling*** at an incremental level, **it supports the modern business practice of**

evolutionary change. By building an architecture that has **extremely small deployment units** that are highly decoupled, architects have a structure that can **support a faster rate of change**.

Performance is often an issue in microservices as it has high performance **overhead**, and they must invoke security **checks to verify identity and access for each endpoint**. *Many patterns exist to increase performance*, including intelligent **data caching** and **replication**. **Performance** is another reason that microservices ***often use*** **choreography** rather than ***orchestration***, as **less coupling** allows for **faster communication and fewer bottlenecks.**

Microservice Decomposition: A Case Study of a Large Industrial Software Migration in the Automotive Industry

Introduction

Challenges and problems of a large monolithic system such as poor maintainability and scaling up and relational databases bottleneck

The rise of microservices as a flexible architecture

Microservices Enhance Maintainability and Scalability but Introduce Complexity and Require Diverse Expertise

Establishing Clear Service Boundaries in Microservices is Essential for Optimal Maintainability and Scalability

Applying Academics Criteria for Modularization like Cohesion and Coupling Falls Short of Practitioner Expectations

Many big companies already have one or more monoliths then need to replace them with microservice architecture but considering doing this in the most efficient way is crucial to make the right decisions.

Related Work

Academics as well as practitioners proposed different approaches to decompose monoliths and agreed on refactoring into microservices architecture.

Decomposing a large monolith system is complicated as you need to understand a big system to know where to draw the border between newly splitted parts and assess the refactoring efforts.

Conway's Law

Software organizations are bound to create system designs which resemble their communication structures.
Explanation : first it is based on just reasoning. The software organizations creating the design are most likely designing it like their structure and communication, if their structure is one large team or large teams they are likely to produce monoliths, else if their structure is small distributed teams then they are more likely to produce distributed architecture.
It is just an observation and the goal is suggesting that having small autonomous teams may lead to more successful implementation of such microservices structure.

a migration can cut along existing boundaries between domain entities following the Bounded Context pattern of Domain Driven design
BCs should be incrementally decreased in size with fewer dependencies.

No Universal Decomposition Approach that works for every project.

Methodology

Utilizing Bounded Context for Monolith Decomposition: Evaluating Impact through Observation and Expert Interviews.

Research objective

Investigating the Effects of Bounded-Context Decomposition and FACADEs Design Pattern on Maintainability and Scalability in Large Industrial Monolith.

What other decomposition approaches promise better results?

Case Study Subject

Transitioning an Automotive Industry's ERP Monolith System to Microservices for Enhanced Scalability and Maintainability as it suffers from poor.

To satisfy the needs of diverse customers from various countries , The system is very configurable including enable/disable components , plugins and integrations.

The decision is to gradually move towards microservices architecture.

Major functional parts about the business of car dealerships are sales and after sales.
There is no formal documented requirements and specifications

One large Subversion SVN was used as a version control system. And used tech stack of Java and spring framework and apache tapestry for frontend.

One major issue was slow development time.
To improve build and development speed , use multiple repositories instead of one big.
Building a large monolith project on a workstation takes 10 minutes.
Continuous Integration builds take 45 mins.

Case Study Design

To check if the approach is well suited to the decomposition , researchers observed the changes by developers to the system for over two years including adding new features as well as modularizing its old monolithic architecture into microservices.
One of the direct measurements was the projects and CI builds time.

Used Decomposition Approach

There were efforts done to reduce the current monolith and split out separate services belonging to the business bounded context. That to increase the speed of the development, Maintainability and scalability.
Teams expected to independently own their part of the whole system. As a result , lines of code were reduced several times through the two years.

Moving existing functionality out of the monolith

First take the business considerations into account , Architects managed to identify a bounded context that is suitable to move into a new microservice.
To make migration easier, the BC shouldn’t be central with high interactions and dependencies in the monolith means a BC incoming but least outgoing calls to other services.
Steps of the migration: let's assume 3 services (a , b , c) , a calls b, b calls c, and only service b is identified as in the bounded context.

Definition of facades:

Generally a facade hides the complexity of internal matters behind a clean interface and system should only be accessed only from these facades
Developers followed architects and created modules with desired communication through rest interfaces calls (APIS) and DTOs.

Facade Implementation

The second major step in the migration and splitting process is implementing the facades as rest services.

Using the facades

The third major step is using the facades of the splitted parts of a monolith in the form of rest endpoints and clients.
The direct dependencies and usages of the new parts about to be splitted like in memory calls have to be replaced across the new border with Rest facade calls.

Transactions become more complex as distribution is not desirable so in many cases it is possible to move the whole transaction to one service and to provide aggregated data from other services.

Visualised case also assumes that the database can be cleanly partitioned between the new microservice and the remaining monolith.

Moving out the separated service part

The final part of the migration and separation process is introducing the new microservice. Developers created two separate git repositories, one for API and another for the service implementation as public and private code and data structures are cleanly separated in that way.

The new microservice application is built on jenkins,sonarqube and the artifacts are deployed. Docker images are built and Helm charts configure how the service should run in a cluster. For the new microservice , the url changed and is no longer part of the monolith.

Rewriting part of the monolith as a microservice

Particularly Problematic pieces with bad maintainability lend not to be moved out to their own microservice BUT to be replaced by a completely new microservice.

Facade introduction and implementation

Set of services should be removed , let's assume this is service b1, service b2 as microservice provides similar functionalities and is destined to replace service b1. See picture.

First step adding facade E around the service about to be replaced and the first implementation delegates to the existing implementation as legacy service b1.Adding Facade F Rest endpoint around the new microservice b2.

Rest client facade implementation

New microservice is not expected to immediately have all functionalities so the migration process takes some time and during that time, Spring profiles are used to switch between

legacy monolithic service implementation with facade E and new microservice implementation with facade F while progressing the development of the new microservice.
In the end, developers should remove the legacy implementation service b1 as new microservice b2 implemented completely and now independent from spring profiles.

Results

Experts confirmed that the chosen approach is indeed well suitable to improve the situation
Although development and migration progress was slow as limited resources were assigned for that.
During the modularization, a lot of files are changed which leads to hard to solve merge conflicts.
Developers perceived working with new small services very positively and the main reasons are easier understandability , greatly improved development speed and faster feedback cycles.These qualities
have a large positive impact on maintainability.
Working with a smaller service also allows to upgrade the code more easily because of fewer dependencies. This leads in the long term for more modern application that developers are satisfied with.
Testers also perspective the microservice approach as preferable.
Test automation is simpler as services can be tested in a more isolated way than the monolith.

Maintainability

Situation improved but still there is much to do.
Smaller modules are easier to understand, start and debug. Testing a smaller application also becomes easier and more efficient.
Deployment speed is also an important part for testers to examine the changes faster.
A negative aspect mentioned is that some actions like version updates have to be repeated for many applications instead of only one monolith.
Developers get productive faster and this in turn improves maintainability.
Debugging becomes harder compared to a monolith where it is possible to debug across the whole application inside an IDE.
For backwards compatible changes a new API version has to be released in addition to the code changes.
As all applications depending on the functionality have to upgrade at the same time or API versions
have to be used. The effort is multiplied by the number of usages.
For this reason it is crucial to keep the number of API changes minimal.

Scalability

Theoretically once the modularization is finished it should allow to scale all applications very well. but it is not experienced in the project yet as the system overall is perceived as rather slow regardless of the load and more replicas do not help directly with latency problems.
As an advantage, a new microservice or other parts under particular high stress , can be scaled independently.
While the scalability of a single service in isolation becomes easier, the whole orchestration becomes more complicated.
Container platforms and other operational details are therefore crucial for the success of the scalability improvements.

END of the case study

Choosing the Appropriate Architecture Style

Choosing an architecture style represents the *peak of analysis and thought about trade-offs* for architecture characteristics, domain considerations, strategic goals, and various other factors.
It depends! With all the choices available (and new ones arriving almost daily), we cannot tell you which one to use.
However **contextual** the decision is, **some general advice exists** around choosing an appropriate architecture style.

Decision Criteria

When choosing an architectural style, an architect must take into account all the various factors that contribute to the structure for the domain design. Fundamentally, an architect designs two things: whatever **domain** has been specified, and all the ***other structural elements required*** to make the system a success.

Architects should go into the design decision **comfortable with the following things**:

The domain

Architects should **understand many important aspects of the domain**, especially those that **affect operational architecture characteristics**.

Architecture characteristics that impact structure

Architects must **discover and clarify the architecture characteristics** needed to support the **domain**.

Data architecture

Architects and DBAs must collaborate on database, schema, and other data-related concerns, they **must understand the impact that data design might have on their design**, particularly if the new system must interact with an older and/or in-use data architecture.

Organizational factors

Many external factors may influence design. For example, **the cost** of a particular cloud vendor may prevent the ideal design. Or perhaps encourages an architect to gravitate toward **open solutions and integration architectures**.

Domain/architecture **isomorphism**

Some problem domains match the topology of the architecture. For example, the **microkernel** architecture style is **perfectly** suited to a system that requires **customizability** (plug-ins).
Similarly, some problem domains may be particularly **ill-suited** for some architecture styles. For example, **highly scalable systems struggle with large monolithic designs** because architects find it difficult to support a large number of concurrent users in **a highly coupled code base**. For instance, an insurance company application consisting of multipage forms, each of which is based on the context of previous pages, would be difficult to model in microservices. This is a highly coupled problem that will present architects with design challenges in a decoupled architecture; **a less coupled architecture like service-based architecture would suit this problem better.**

Taking all these things into account, the architect must make several determinations:

- **Monolith versus distributed**

Using the **quantum concepts** discussed earlier, the architect must determine if a **single set** of architecture **characteristics will suffice** for the design, or do **different parts need differing architecture characteristics? A single set** often implies that a **monolith** is suitable, whereas **different** architecture characteristics often imply a **distributed architecture**.

- **Where should data live?**
  If the architecture is **monolithic**, architects commonly assume **a single relational database (with replicas)**. In a **distributed** architecture, the architect **must decide which services should persist data**. Architects must consider both **structure and behavior** when designing architecture and ***to Iterate on the design to find better combinations.***

- **What communication styles between services—synchronous or asynchronous? synchronous or asynchronous? Synchronous communication is more convenient in most cases**, but it can lead to *lack of scalability, reliability*, and other desirable characteristics. **Asynchronous** communication can provide **unique benefits** in terms of **performance and scale** but can present a host of headaches: *data synchronization, deadlocks, race conditions, debugging*, and so on.

Because synchronous communication presents fewer design, implementation, and debugging challenges, architects should **default to synchronous** when possible and **use asynchronous when necessary**.

Use synchronous by default, asynchronous when necessary.

The output of this design process is **architecture topology**, the architecture style chosen and **architecture fitness functions** to **protect important principles** and operational architecture **characteristics**.

Diagramming and Presenting Architecture

Effective communication becomes critical to an architect's success. No matter how brilliant an architect's technical ideas, **if they can't convince managers to fund them and developers to build them*, their brilliance will never manifest.***

When visually describing an architecture, the architect will likely **show an overview** of the entire architecture topology, then **drill into individual parts** to delve into design details. However, **if**

the architect shows a portion without indicating where it lies within the overall architecture, *it confuses viewers*. **Representational consistency** is the **practice** of always **showing the relationship between parts of an architecture**, either in diagrams or presentations**, before changing views.**

Diagramming

The topology of architecture captures how the structure fits together and forms a valuable shared understanding across the team. Therefore, **architects** should hone their **diagramming skills** to razor sharpness.

Irrational Artifact Attachment anti-pattern

As an architect, it's crucial to approach your design process with **flexibility and openness to change**. Early in your **design phase**, **opt for low-fidelity artifacts**—*think index cards, sticky notes, or simple sketches.* These tools are not only quick and inexpensive to use, but they also **help you avoid becoming overly attached to your initial ideas**, **a common pitfall known as the Irrational Artifact Attachment anti-pattern.**

The more time you invest in creating a beautiful detailed diagram, can lead to a disproportionate attachment to it. This attachment might make it **difficult for you to accept necessary changes.** For instance, if you **spend several hours** perfecting a diagram in a high-fidelity tool like Visio, you **might find yourself resistant to altering your design based on feedback or new insights.**

To stay agile and responsive, embrace the practice of **creating just-in-time artifacts** with minimal ceremony. This approach aligns well with Agile methodologies and keeps your process dynamic. **By using simpler, easily modifiable tools** early on, **you give yourself the freedom to experiment and iterate with the team members.**

Tools

Eventually, an architect needs to create nice diagrams in a fancy tool, but make sure the team has iterated on the design sufficiently to invest time in capturing something.

Powerful tools exist to create diagrams on every platform and while we don’t necessarily advocate one over another,
here is a suggested list of tools that are sufficient for most of the diagramming needs:

- Draw.io (Free online Visio Alternative)
- Microsoft Visio
- OmniGraffle : Preferred by Mac users, known for its user-friendly interface and powerful capabilities.

- Diagrams: Show Me (on ChatGPT plus)

The “Diagrams: Show Me” Custom ChatGPT is a really great and impressive **AI tool** that I used recently.
Simply describe the diagram you need with all details, and the tool will generate it for you. This can be useful for **architects who need to quickly visualize concepts**. Once created, diagrams can be edited and downloaded in various formats directly from the tool’s website.

Diagramming Standards: UML and C4

Several **formal standards** exist for **technical diagrams** in software.

UML

Unified Modeling Language (UML) **was a standard** that unified three competing design philosophies that coexisted in the 1980s.
Architects and developers still use UML class and sequence diagrams to communicate structure and workflow, *but most of the other UML diagram types have fallen into disuse.*

C4

C4 is a diagramming technique developed by Simon Brown **to address deficiencies in UML and modernize its approach**.

The four C’s in C4 are as follows:

- **Context**
  Represents the entire context of the system, including the roles of users and external dependencies.

- **Container**
  The physical (and often logical) **deployment boundaries** and **containers** within the architecture. This view forms a good meeting point for operations and architects.

- **Component**
  The component view of the system; this most neatly **aligns with an architect’s view of the system**.

- **Class**

> C4 uses the **same style of class diagrams from UML**, which are effective, so there is no need to replace them.

If a company seeks to standardize on a diagramming technique, C4 offers a good alternative. However, like all technical diagramming tools, **it suffers from an inability to express every kind of design an architecture might undertake.**
C4 is best suited for monolithic architectures where the container and component relationships may differ, **and it's less suited to distributed architectures, such as microservices**.

Presenting

The other soft skill required by modern architects is the ability to conduct effective presentations using tools like PowerPoint and Keynote.
In a presentation, the presenter controls **how quickly an idea is unfolding rather than a reader reading a document**. **Thus, one of the most important skills an architect can learn** in their presentation tool of choice is **how to manipulate time.**

Manipulating Time

Designers shouldn't artificially pad content to make it appear to fill a slide as many ideas are bigger than a single slide.
Presentation tools offer two ways to manipulate time on slides: **transitions and animations**. Transitions move from slide to slide, and animations allow the designer to create movement within a slide.
Using subtle combinations of transitions and animations allows presenters **to separate ideas to individual slides and stitch together a set of slides to tell a single story**. **To indicate the end of a thought, presenters should use a distinctly different transition.**

Most readers have the **excruciating experience of watching a slide full of text appear during a presentation**, only to sit for the next 10 minutes while the presenter **slowly reads the same text and bullets to the audience**. No wonder so many corporate presentations are dull! The **better solution** to this problem is **to use incremental builds for slides**, building up (hopefully graphical) information as needed rather than all at once per slide.

Invisibility is also a simple effective pattern where the presenter **inserts a blank black slide** within a presentation **to refocus attention solely on the speaker**. **If a presenter wants to make a point**, insert a blank slide—**everyone in the room will focus their attention back on the speaker.**

Learning the basics of a presentation tool and a few techniques to make presentations better **is a great addition to the skill set of architects.** If an architect has a great idea but **can't figure out a way to present it effectively, they will never get a chance to realize that vision.** Architecture requires collaboration; **to get collaborators, architects must convince people to sign on to their vision**.

Developing a Career Path

Becoming an architect takes time and effort, but based on the many reasons we've outlined before, **managing a career path after becoming an architect is equally tricky.** While we can't chart a specific career path for you,
we can point you to some practices that we have seen work well.

An architect must continue to learn throughout their career. The technology world changes at a dizzying pace. **Each architect should keep an eye out for relevant resources, both technology and business.** Unfortunately, **resources come and go all too quickly**, which is why we don't list any. **Talking to colleagues or experts about what resources they use** to keep current is one good way of seeking out the latest newsfeeds, websites, and groups that are active in a particular area of interest.

Architects should also build into their day some time

to maintain breadth utilizing those resources.

20-minute rule

Technology breadth is more important to architects than depth. However, maintaining breadth takes time and effort, something **architects should build into their day**. But how in the world does anyone have the time to actually go to various websites to read articles, watch presentations, and listen to podcasts? The answer is…not many do. **Developers and architects alike struggle with the balance of working a regular job,** spending time with the family, being available for our children, carving out personal time for interests and hobbies, and trying to develop careers, while at the same time trying to keep up with the latest trends and buzzwords.

One technique we use to maintain this balance is something we call the 20-minute rule. The idea of this technique is to **devote at least 20 minutes a day to your career as an architect by learning something new or diving deeper into a specific topic**.
Spend that minimum of 20 minutes to Google some unfamiliar buzzwords ("the things you don't know you don't know") to learn a little about them, promoting that knowledge into the "things you

know you don't know." Or maybe spend the 20 minutes going deeper into a particular topic to gain a little more knowledge about it. The point of this technique is to be able to carve out some time for developing a career as an architect and **continuously gaining technical breadth.**

We strongly recommend leveraging the 20-minute rule first thing in the morning, as the day is starting.

We recommend a list of resources such as websites that may benefit you in gaining technical breadth using the 20 minutes rule:

- InfoQ.com
- DZone.com/refcardz
- ThoughtWorks.com/radar

Practice is the proven way to build skills and become better at anything in life…including architecture. We encourage new and existing architects to keep honing their skills, both for individual technology breadth but also for the craft of designing architecture.

References:

https://en.wikipedia.org/wiki/Software_architecture

https://www.netsolutions.com/insights/why-software-architecture-matters-to-build-scalable-solutions/#:~:text=Good%20software%20architecture%20helps%20maintain,%2C%20scalable%2C%20and%20reliable%20software.

https://thedomaindrivendesign.io/bounded-context/