Welcome

Working with Design Patterns

Hello

About me...

- ~30 years in the industry
- 25+ years in teaching
- Certified Cloud architect
- Passionate about learning
- Also, passionate about Reese's Cups!

Why study this subject?

- Building software systems is “easy” – building software systems that perform, scale, are testable, and that provide lasting value is not
- Architectural quality does not happen by accident
- Having said that, there are some key principles and practices that help us be successful
- Spending some time laying a foundation in those principles and practices is time well spent

We teach over 400 technology topics

You experience our impact on a daily basis!

My pledge to you

I will...

- Make this interactive
- Ask you questions
- Ensure everyone can speak
- Use an on-screen timer

Objectives

At the end of this course, you will be able to:

- Gain an in-depth understanding of key software architecture concepts and methodologies
- Grasp how to refactor complex systems to achieve test coverage and scalability
- Recognize the tradeoffs between alternate designs depending upon staffing, time to market, and scalability constraints

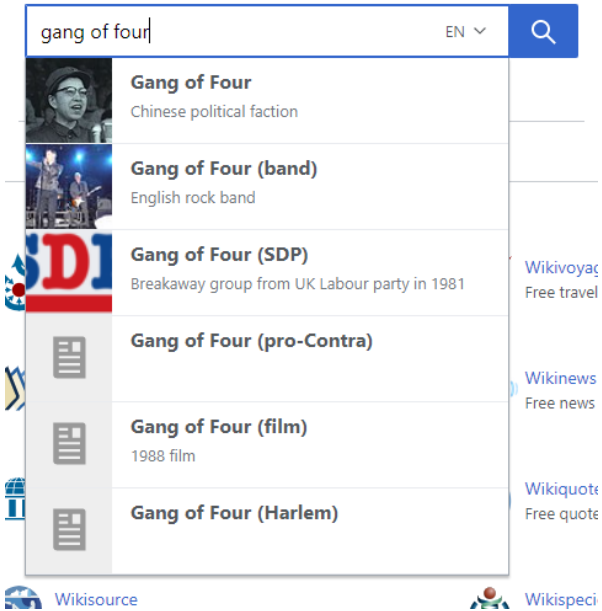
Origin of Design Patterns

First Use of “Design Pattern”

- Term first coined by an architect and anthropologist – Christopher Alexander
- Presented a new language construct based around an entity called a “pattern”
- Pattern describes a problem and provides a reusable (and proven) solution to that problem

Gang of Four (GoF)

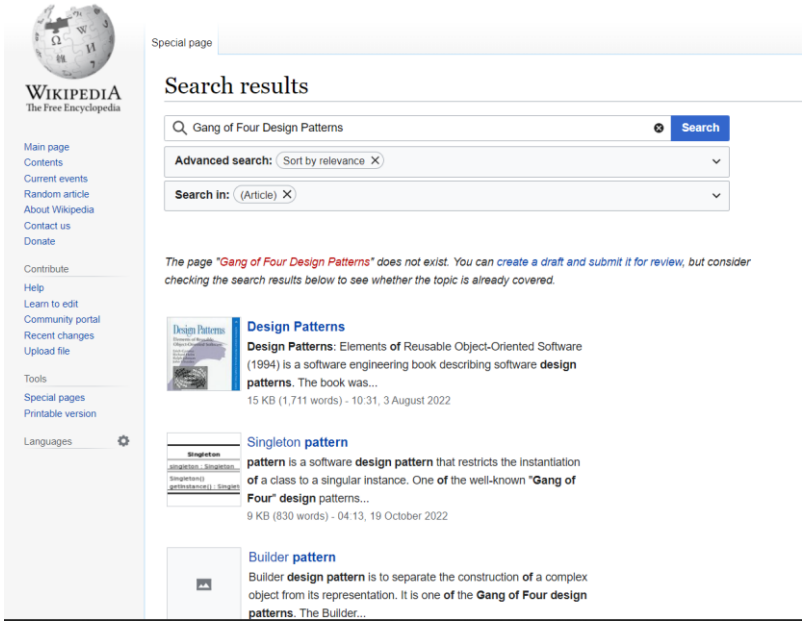
- Initial search for “Gang of Four” on www.wikipedia.org



Not exactly what we’re looking for…

Gang of Four (GoF)

- Search for “Gang of Four Design Patterns”



That’s more like it…

Gang of Four (GoF)

- Group of 4 authors who wrote the book titled “Design Patterns: Elements of Reusable Object-Oriented Software” (1994)
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
- Includes detail on 3 types of patterns

Gang of Four (GoF)

- Creational
- Structural
- Behavioral

Gang of Four (GoF)

Supports creation of objects indirectly (in a more loosely-coupled fashion); enables association of logic to determine what and how to create

- Creational
- Structural
- Behavioral

Gang of Four (GoF)

Creational

Structural

Behavioral

About class and object composition; using inheritance and extension to build out entity hierarchies that match with the “real world” and enable layering in new functionality in an architecturally sound manner

Gang of Four (GoF)

Creational

Structural

Behavioral

Mainly manages concepts of communication between objects – building out a messaging system that allows us to break a larger problem into smaller pieces but still coordinate

Software Architecture Patterns vs. Design Patterns

Architecture Style

Architecture Pattern

Design Pattern

Software Architecture Patterns vs. Design Patterns

Describe the macro structure of a system (e.g., event-driven architecture)

Architecture Style

Architecture Pattern

Design Pattern

Software Architecture Patterns vs. Design Patterns

Architecture Style

Architecture Pattern

Design Pattern

Reusable structural pattern that can be used to enable the architecture style (e.g., CQRS)

Software Architecture Patterns vs. Design Patterns

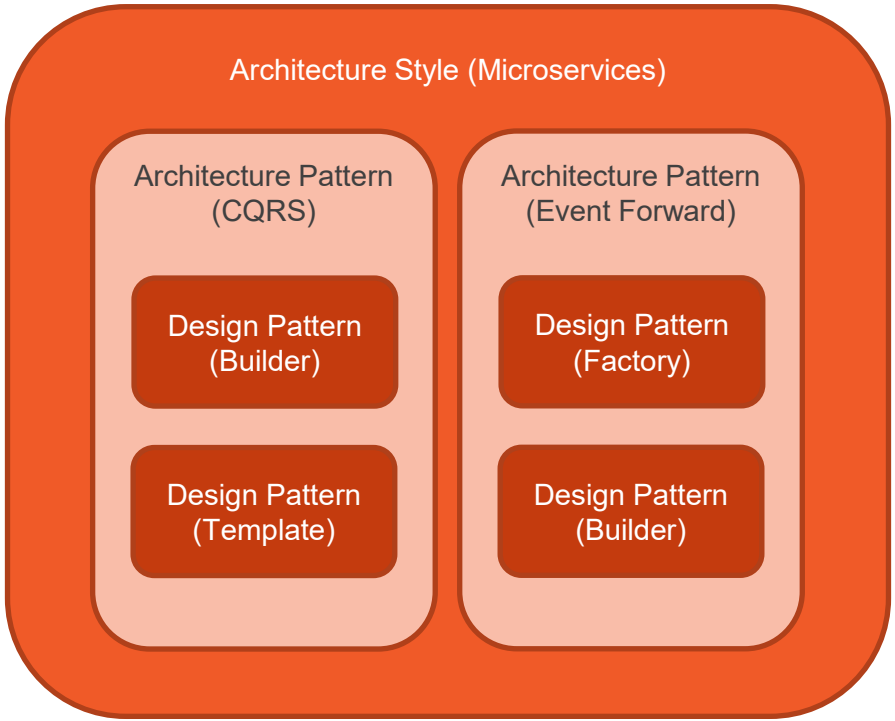
Architecture Style

Architecture Pattern

Design Pattern

Impacts how source code is designed and structured

Architecture Style, Architecture Pattern, Design Pattern



QUICK CHECK

Architecture Patterns & Design Patterns

For the following “boxes”, specify whether you believe the concept represented by the box is an architecture pattern or a design pattern (given the distinction/definitions just discussed).

Layered	Microkernel
Singleton	Observer
Factory	Adapter
Microservices	Chain of Responsibility

Concepts that Enable Robust Software

Architecting for the Future

SOLID Principles

SOLID principles help us build testable code

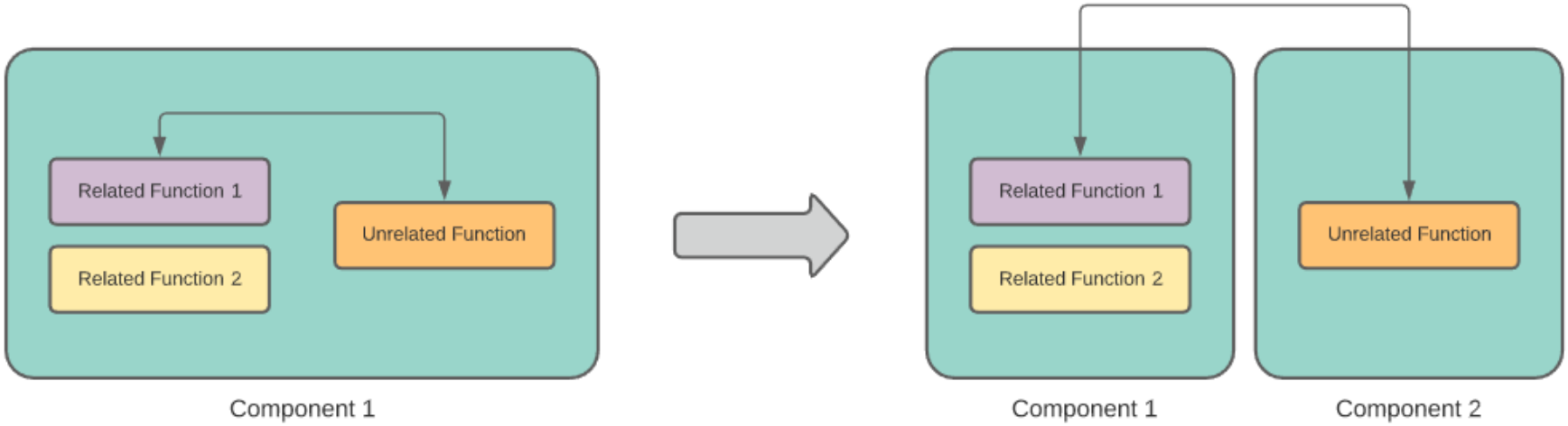
- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Single Responsibility Principle (SRP)

A system module or component should have only one reason to change

Single Responsibility Principle (SRP)

Single Responsibility Principle (SRP)



LAB 01:

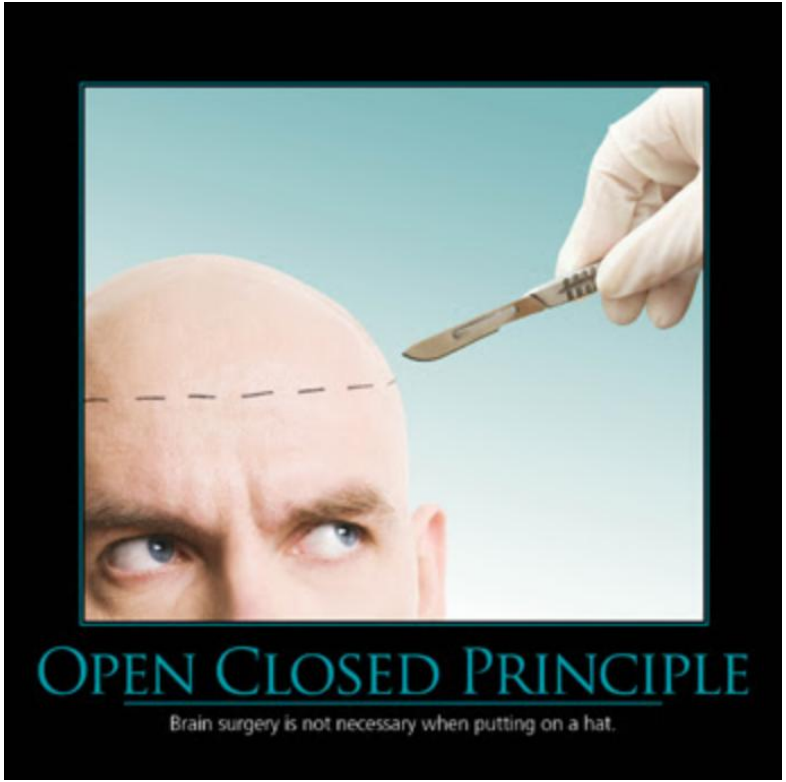
Single Responsibility Principle (SRP)

https://github.com/KernelGamut32/working-with-design-patterns-public/tree/main/labs/lab01

Open-Closed Principle (OCP)

Software entities should be open for extension but closed for modification

Open-Closed Principle (OCP)



LAB 02:

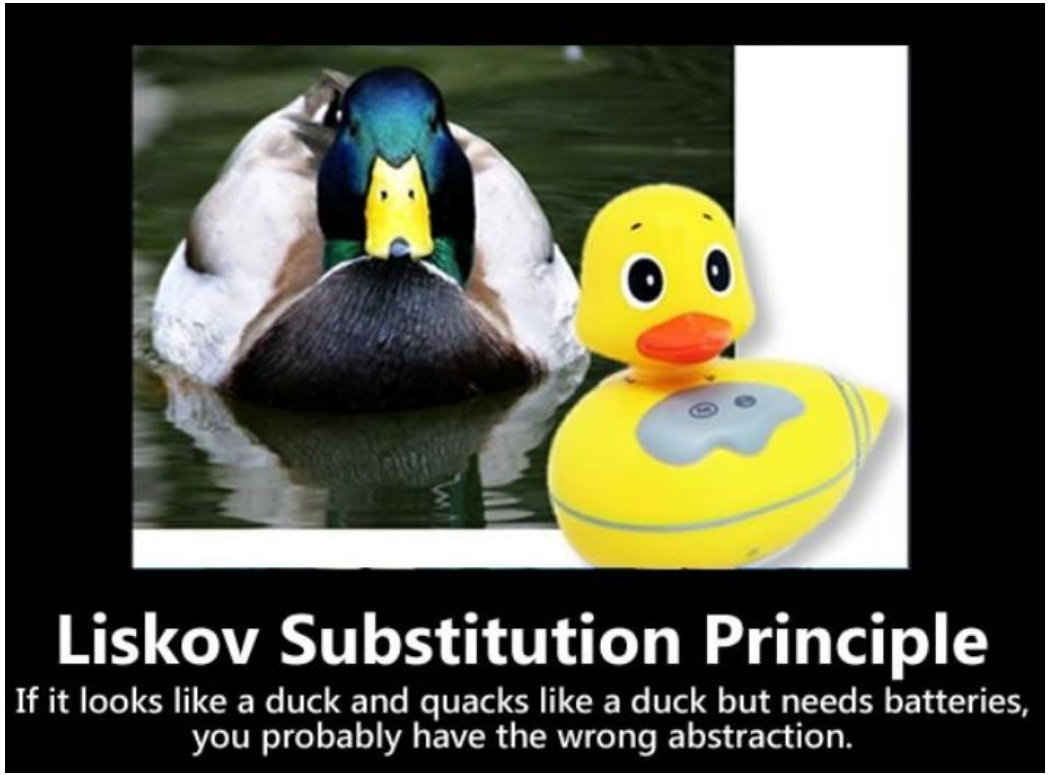
Open Closed Principle (OCP)

https://github.com/KernelGamut32/working-with-design-patterns-public/tree/main/labs/lab02

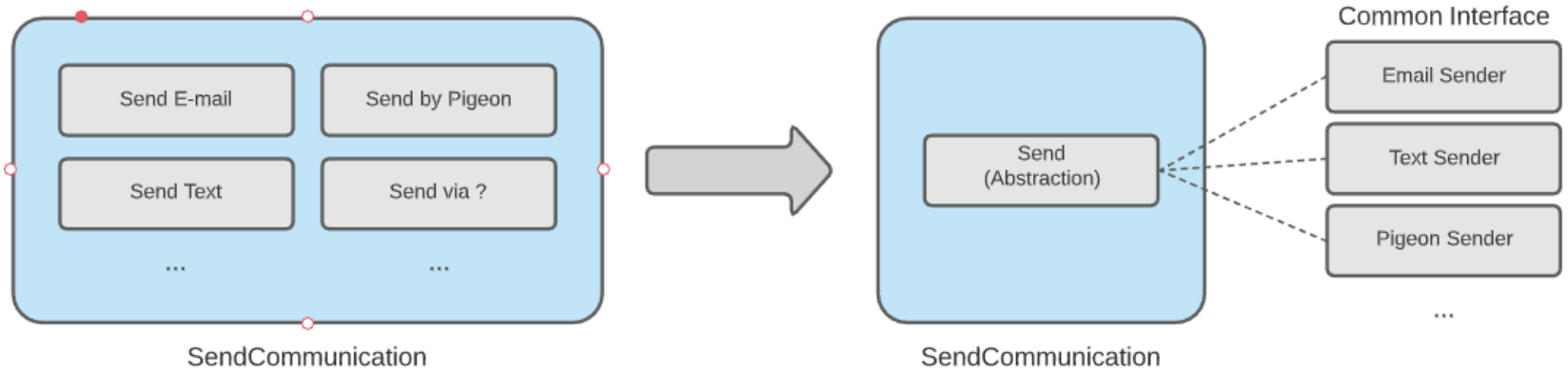
Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types

Liskov Substitution Principle (LSP)



OCP & LSP


Send E-mail
Send by Pigeon
Send Text
Send via ?
...
...
SendCommunication
Send
(Abstraction)
Common Interface
Email Sender
Text Sender
Pigeon Sender
...
SendCommunication

LAB 03:

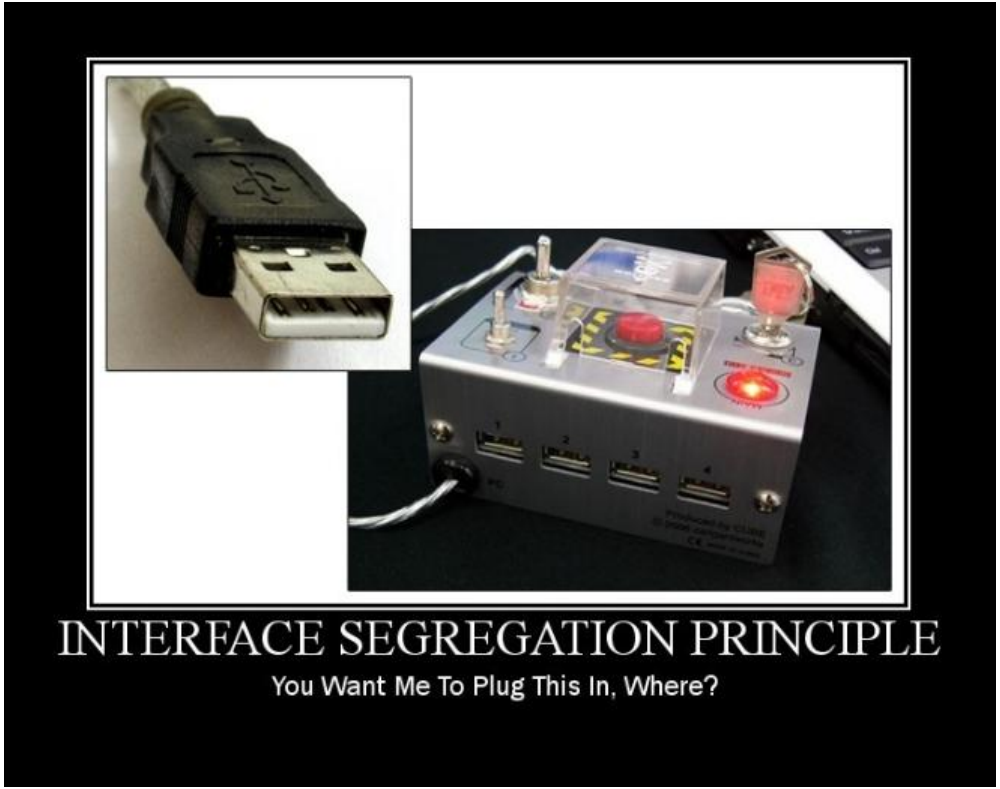
Liskov Substitution Principle (LSP)

https://github.com/KernelGamut32/working-with-design-patterns-public/tree/main/labs/lab03

Interface Segregation Principle (ISP)

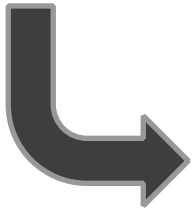
Clients should not be forced to depend on methods they do not use

Interface Segregation Principle (ISP)



Interface Segregation Principle (ISP)

```
7   namespace CoolCompany.Financials
8   {
9       public interface ITaxProcessor
10      {
11          double CalculateSalesTax(double onAmount);
12          double CalculatePropertyTax(double onAmount);
13          double CalculateIncomeTax(double onAmount);
14      }
15  }
```



```
7   namespace CoolCompany.Financials
8   {
9       public interface ITaxProcessor
10      {
11          double Calculate(double onAmount);
12      }
13
14      public interface ISalesTaxProcessor : ITaxProcessor
15      {
16          double LookupStateTaxRate(string state);
17      }
18
19      public interface IPropertyTaxProcessor : ITaxProcessor
20      {
21          double LookupTownshipTaxRate(string township);
22      }
23
24      public interface IIncomeTaxProcessor : ITaxProcessor
25      {
26          double CalculateBackTaxes(string taxpayerId);
27      }
28  }
```

LAB 04:

Interface Segregation Principle (ISP)

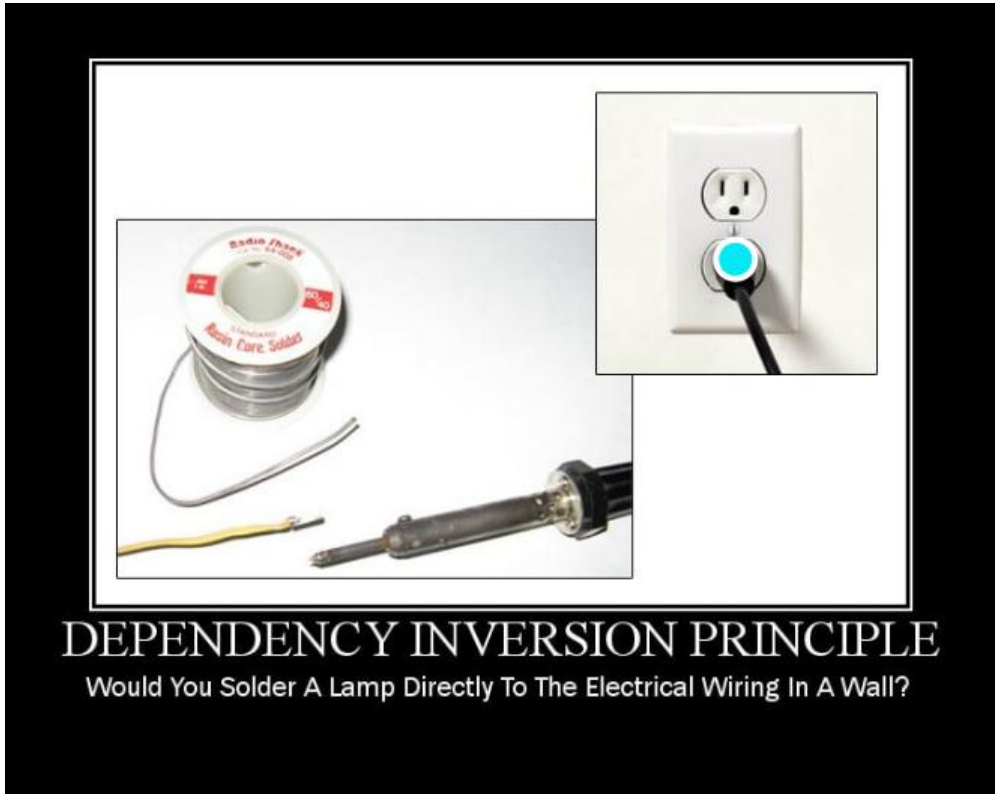
https://github.com/KernelGamut32/working-with-design-patterns-public/tree/main/labs/lab04

Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules – both should depend on abstractions

Abstractions should not depend upon details – details should depend upon abstractions

Dependency Inversion Principle (DIP)



LAB 05:

Dependency Inversion Principle (DIP)

https://github.com/KernelGamut32/working-with-design-patterns-public/tree/main/labs/lab05

Test-Driven Development (TDD)

Benefits to be experienced with automated testing

Greater shared success
(as tests can be tied to requirements)

Cleaner code

Guaranteed system documentation

Improved quality

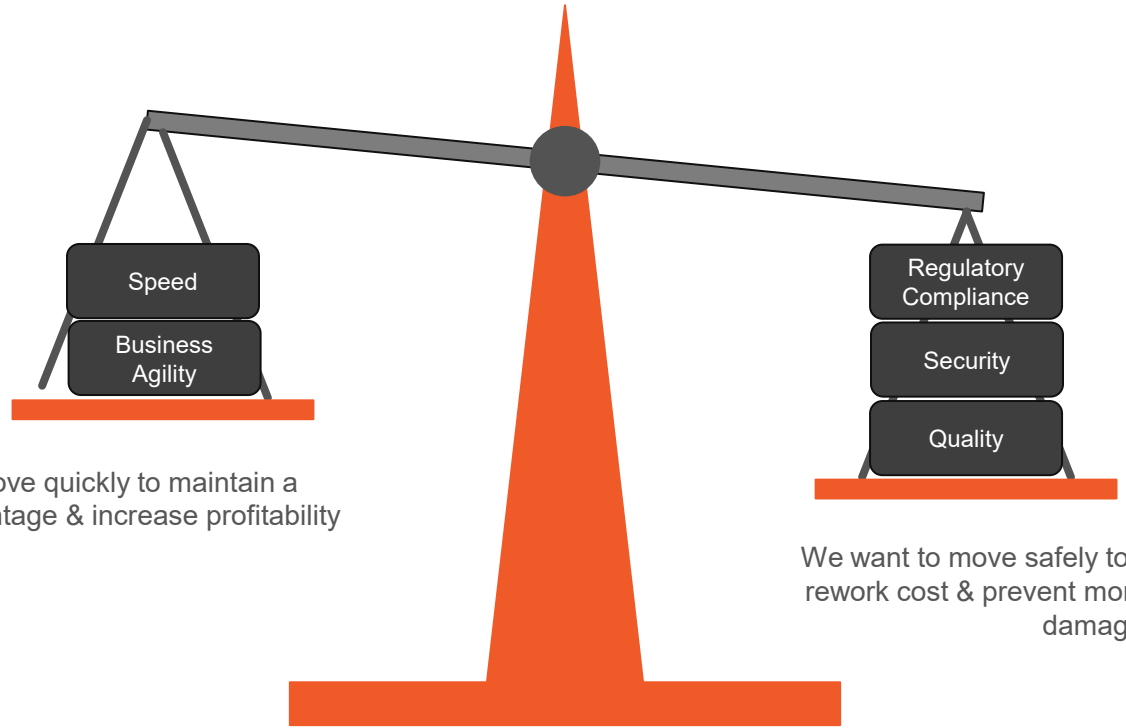
Reduced fear

Improved regression protection

Greater streamlining to the approach
(small bites, avoidance of premature optimization, structured flow)

Discussion | Which of these benefits resonate most with you?

Balancing Speed & Quality



We want to move quickly to maintain a competitive advantage & increase profitability

We want to move safely to ensure we minimize rework cost & prevent monetary & reputational damage

Test-Driven Development (TDD) and the 3 Laws

1

The First Law

"You are not allowed to write any production code until you have written a unit test that fails due to its absence"

2

The Second Law

"You are not allowed to write more of a unit test than is sufficient to fail, and failing to compile is failing"

3

The Third Law

"You are not allowed to write more production code than is sufficient to cause the currently failing test to pass"

Understanding TDD

Scenario: Determine discount based on purchase volume

- SaaS module is external service defining and enforcing business rules
- Several levels of discount based on purchase volume ranges
- Need to verify that the invoicing module operates correctly with the different levels of discount
- What if the call to the SaaS module fails?

- Tie test cases directly to specific business requirements (e.g., each discount level)
- Write just enough code to pass the tests
- By extension, have written just enough code to meet the business requirements
- Mocking enables tight control over what gets returned from the downstream dependency, including any error conditions
- Result is verified quality when built and automated regression testing against future changes

Best practices with TDD

TDD supports the following best practices:

Don’t commit to a methodology, commit to impact and a positive result

Ensure that the engineering team has a clear understanding (and practice) for unit vs. integration testing

Automated test code is code – just like application code – estimate and plan accordingly

Find the right % of test coverage – quality & quantity!

To be effective, automated unit tests need to follow FIRST:

F

Fast

or they won't be executed

I

Independent

i.e., minimal dependencies between tests (keeps them simpler)

R

Repeatable

i.e., every run against unchanged code should operate the same

S

Self-validating

i.e., Boolean output (pass or fail)

T

Timely

if not first, early and often

Challenges

What are some of the challenges you face with software testing in your jobs (i.e., why don’t we test as much as we want to or should)?

Challenges

- Time pressures
- Complexity
- Requires more code
- More debugging (up front)
- More cost (up front)
- “Can’t we just ensure quality with other types of testing?”

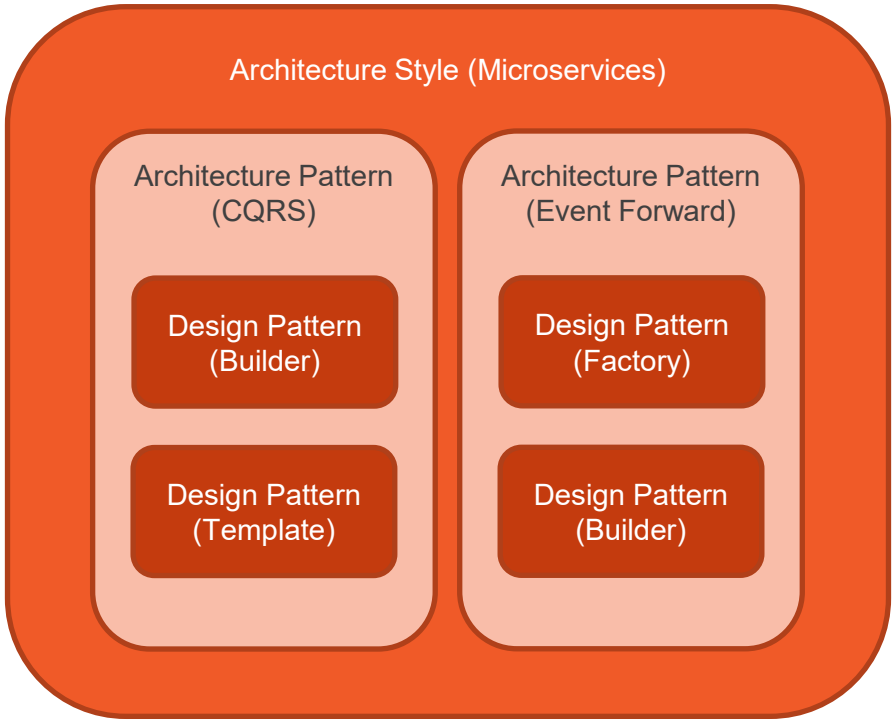
LAB 06:

TDD

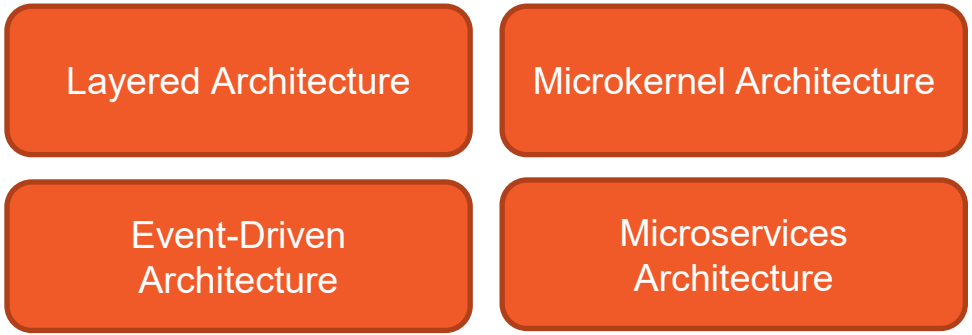
https://github.com/KernelGamut32/working-with-design-patterns-public/tree/main/labs/lab06

Software Architecture Styles

Redux


Architecture Style (Microservices)
Architecture Pattern (CQRS)
Design Pattern (Builder)
Design Pattern (Template)
Architecture Pattern (Event Forward)
Design Pattern (Factory)
Design Pattern (Builder)

Common Software Architecture Styles



Technical vs. Domain Partitioning

Technical

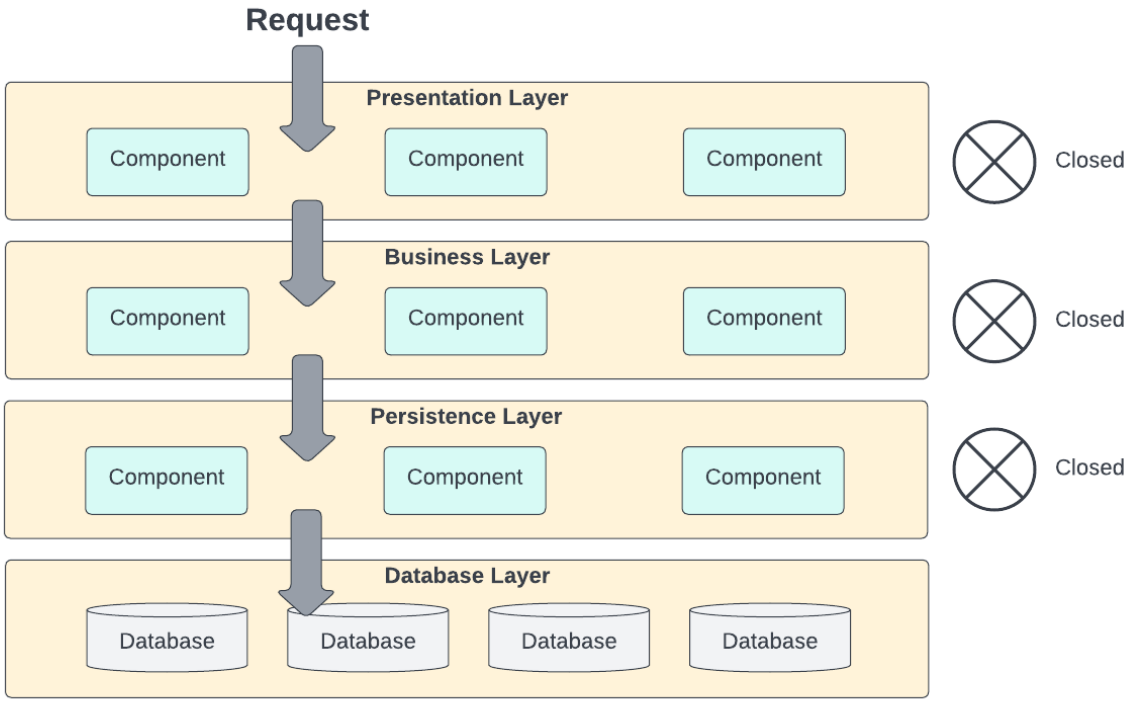
- Components organized by technical usage
- Elements of a given business domain are spread across multiple components or layers
- Useful if change isolated to a specific technical area
- Includes Layered, Microkernel, Event-Driven, and Space-Based

Domain

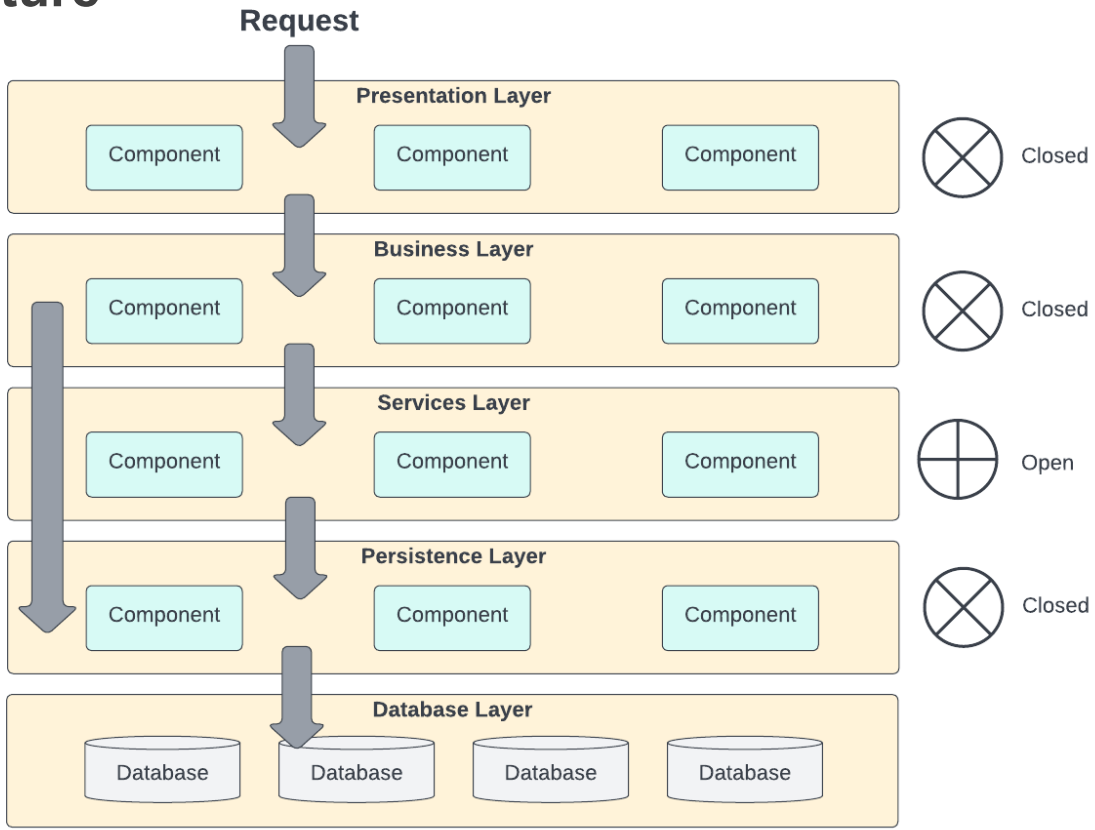
- Organized by business domain area vs. technical usage
- AKA Domain-Driven Design (DDD)
- Changes typically isolated to an individual domain
- Helps align technical implementation with business processes
- Includes Microkernel and Microservices

Layered Architecture

Layered Architecture



Layered Architecture



Group Discussion:

Layered Architecture

As a group, discuss the pros & cons of this architectural style. Specifically:

- What are some advantages to using this style? Is there a specific application profile that would cause you to prefer this style?
- What are some disadvantages to using this style? Is there a specific application profile that would you to avoid this style?

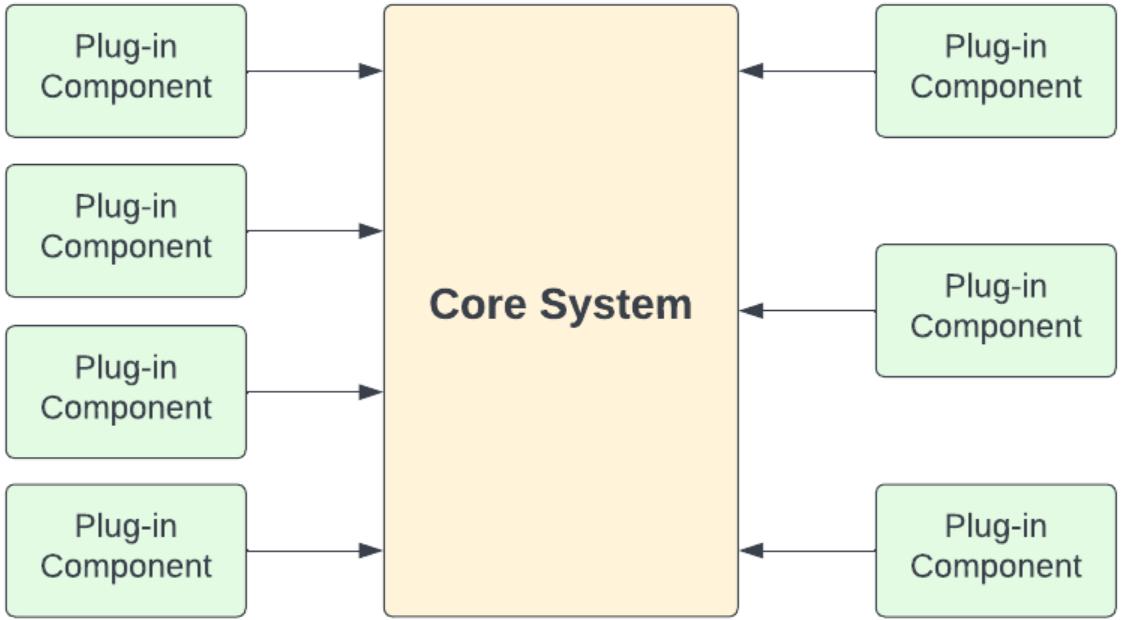
SOLID in Practice

https://github.com/KernelGamut32/working-with-design-patterns-public/blob/main/use-cases/Test%20Driven%20Development%20(TDD)%20-%20Class%20Project.pdf

Microkernel Architecture

Microkernel Architecture

Independent, standalone modules containing specialized logic to address one or more specific use cases or provide specialized business capability



Minimal functionality required to make base system operational

Microkernel Architecture

Key Components & Strategies of Approach

Keep cross-plugin communication to a minimum

Core system uses a separate plug-in registry to know what's registered and available

Implemented as separate libraries or modules (can also be implemented remotely)

Group Discussion:

Microkernel Architecture

As a group, discuss the pros & cons of this architectural style. Specifically:

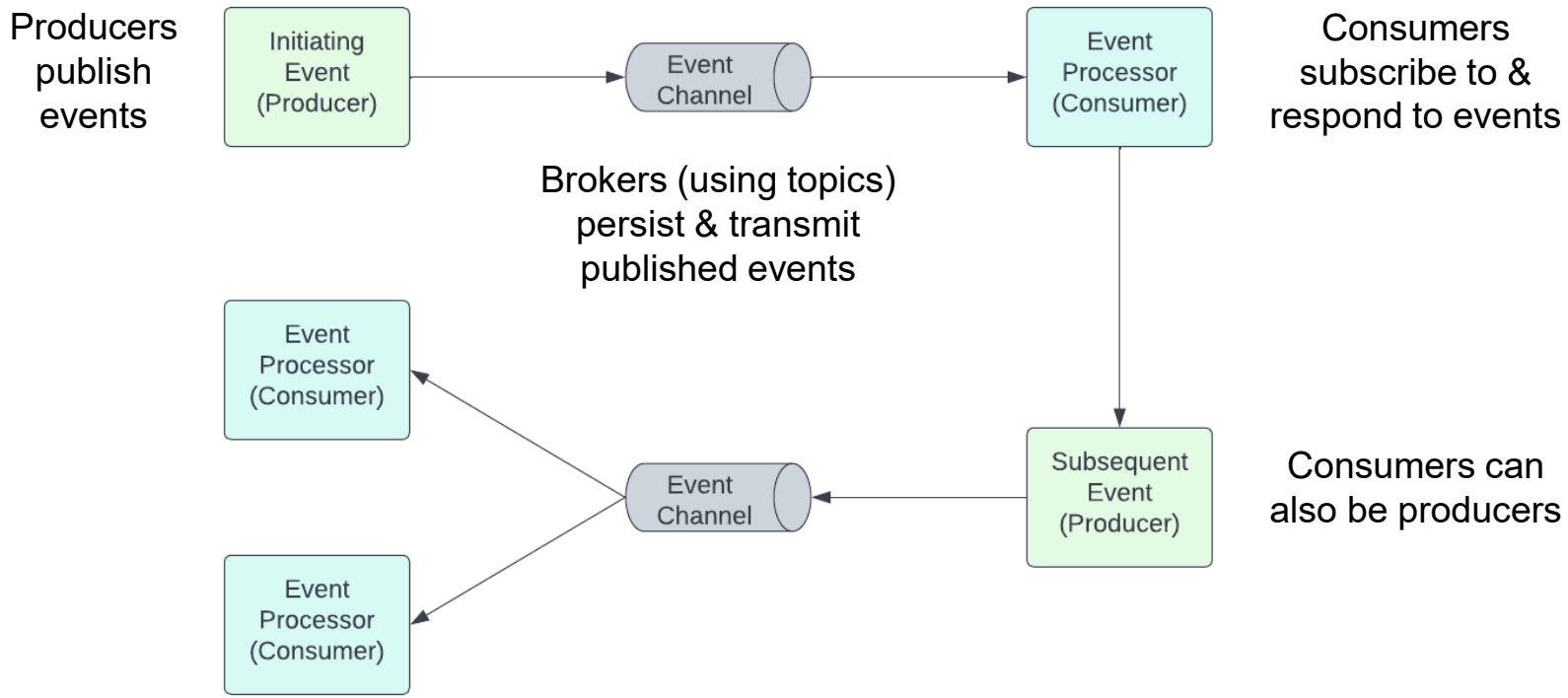
- What are some advantages to using this style? Is there a specific application profile that would cause you to prefer this style?
- What are some disadvantages to using this style? Is there a specific application profile that would you to avoid this style?

Follow the Rules

https://github.com/KernelGamut32/working-with-design-patterns-public/tree/main/use-cases/rules-engine

Event-Driven Architecture

Event-Driven Architecture

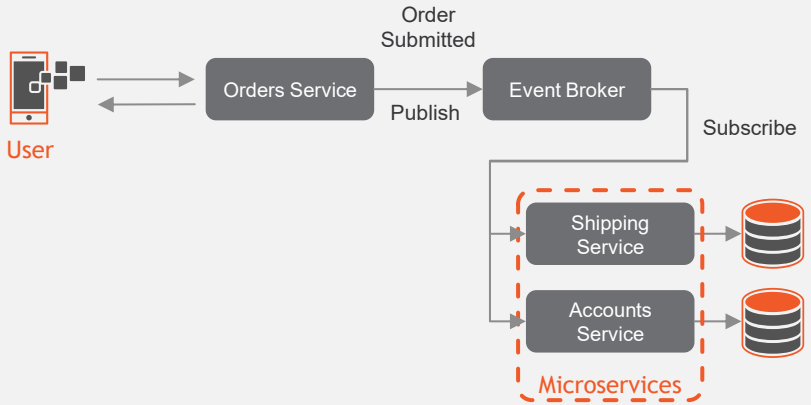


Event-Driven Architectures

As an architectural style:

- Enables looser coupling between communicating services
- Provides dynamic flexibility in the definition of integrated workflows
- Promotes resiliency as an intermediate component (the broker) is used to separately track and manage the queue events
- Uses concept of domain events – publish of events with business significance

Simplified example

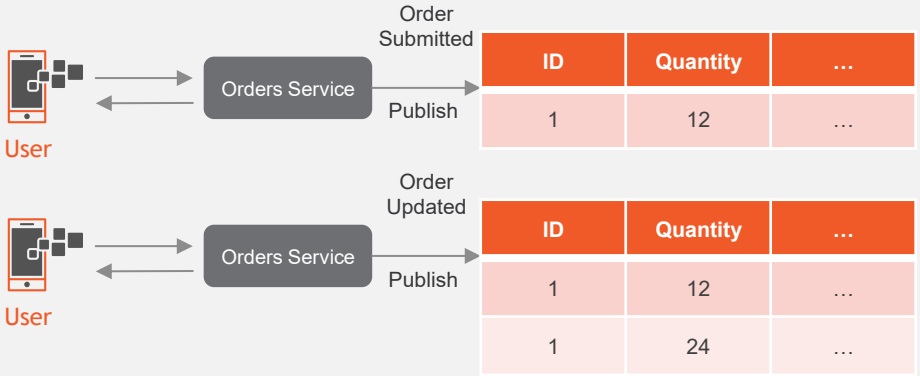


Event-Driven Architectures

Event Sourcing

- Data changes are captured as immutable events
- When a change is made, rather than overwrite existing record, a new record is created to reflect updated state
- Provides a full picture of what happened during an entity's lifecycle – no data destruction
- Can make reads more difficult

Simplified example

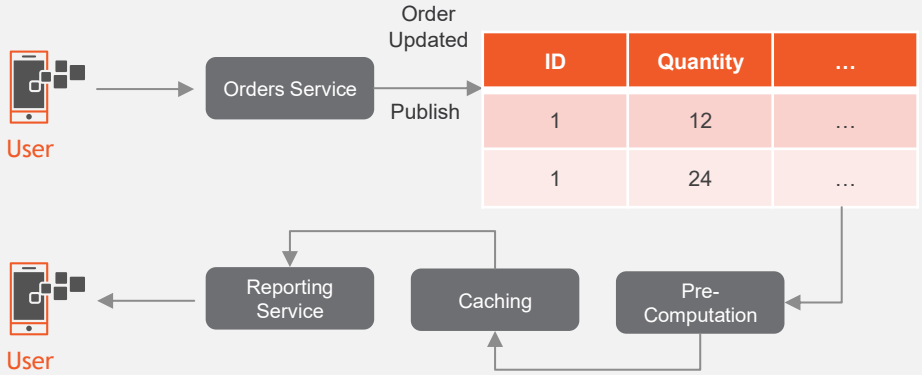


Event-Driven Architectures

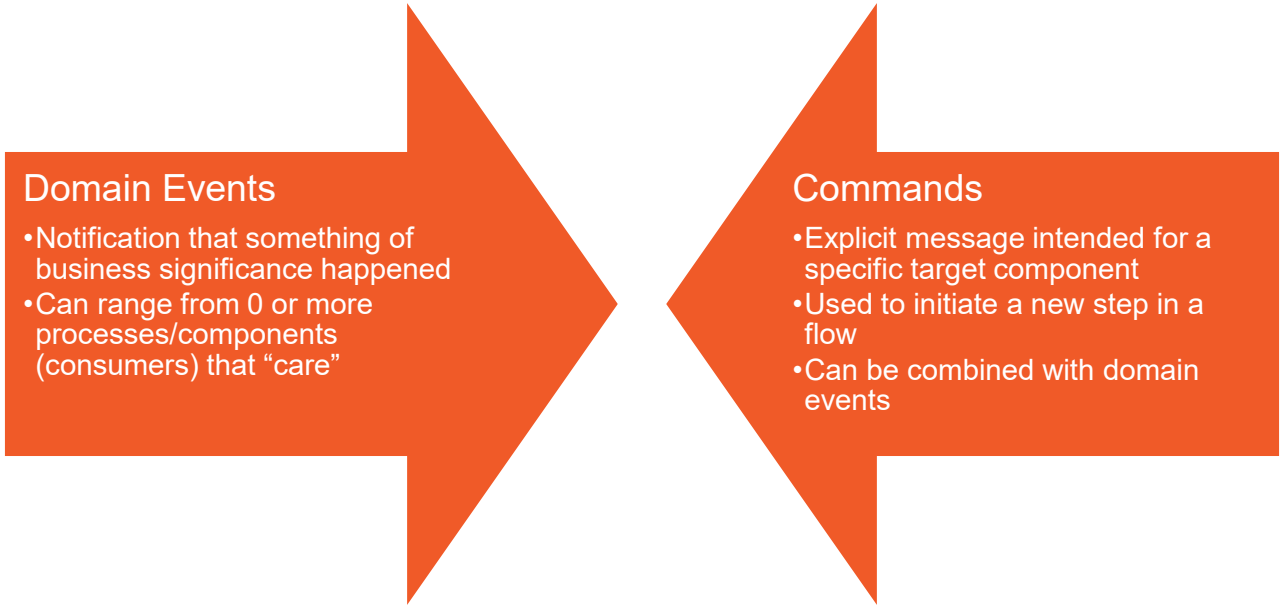
CQRS

- Command-Query Responsibility Segregation
- Can help address challenges with Event Sourcing architectures
- Separate processes used for writes and reads
- Allows write optimization (“raw” events) and read optimization (dedicated aggregation)
- Also, enables denormalization and reformatting for reads separate from write structures

Simplified example



Domain Events vs. Commands (or Messages)



Group Discussion:

Event-Driven Architecture

As a group, discuss the pros & cons of this architectural style. Specifically:

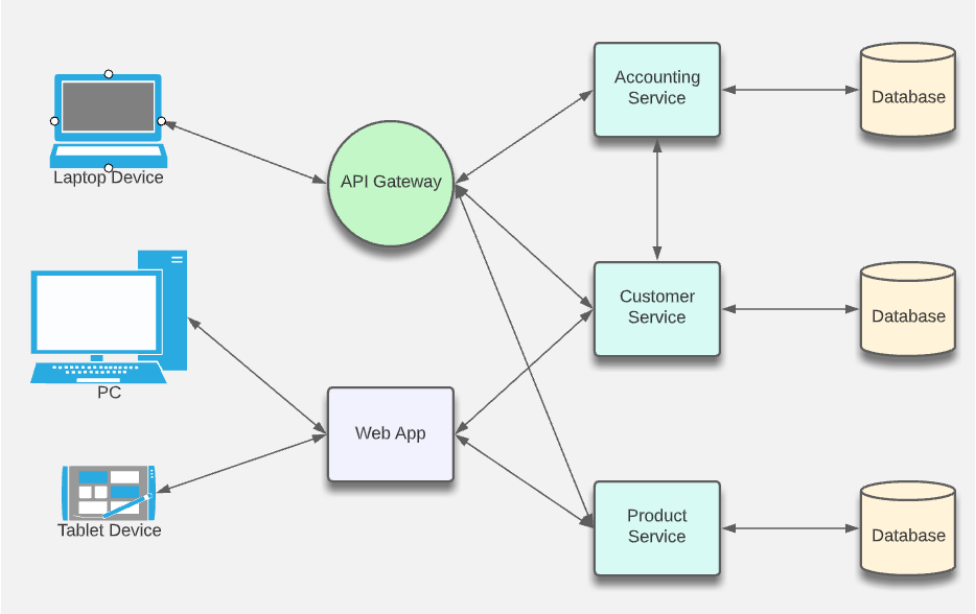
- What are some advantages to using this style? Is there a specific application profile that would cause you to prefer this style?
- What are some disadvantages to using this style? Is there a specific application profile that would you to avoid this style?

Vacationing on Lake Data

https://github.com/KernelGamut32/working-with-design-patterns-public/tree/main/use-cases/data-mart

Microservices Architecture

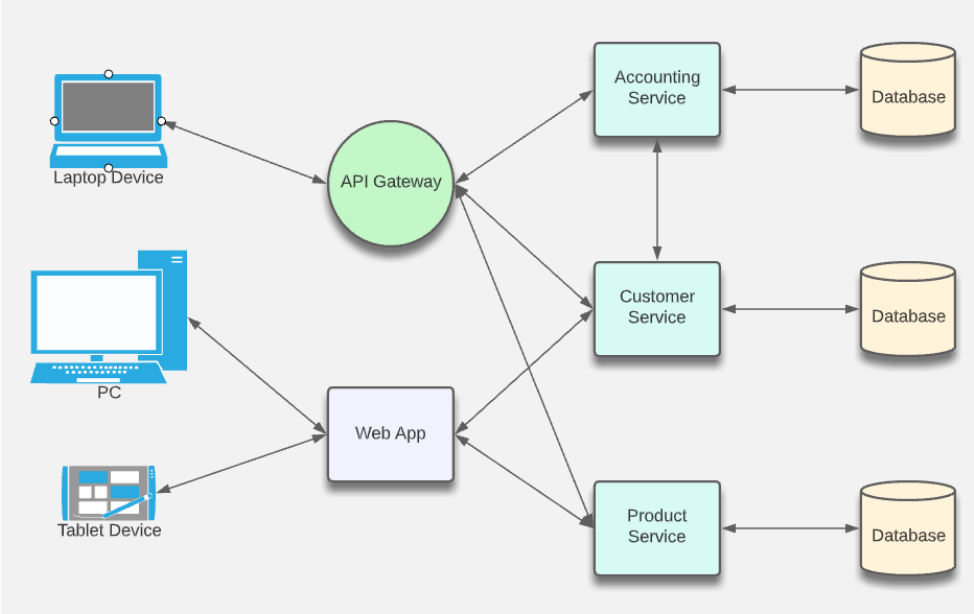
Microservices Architecture



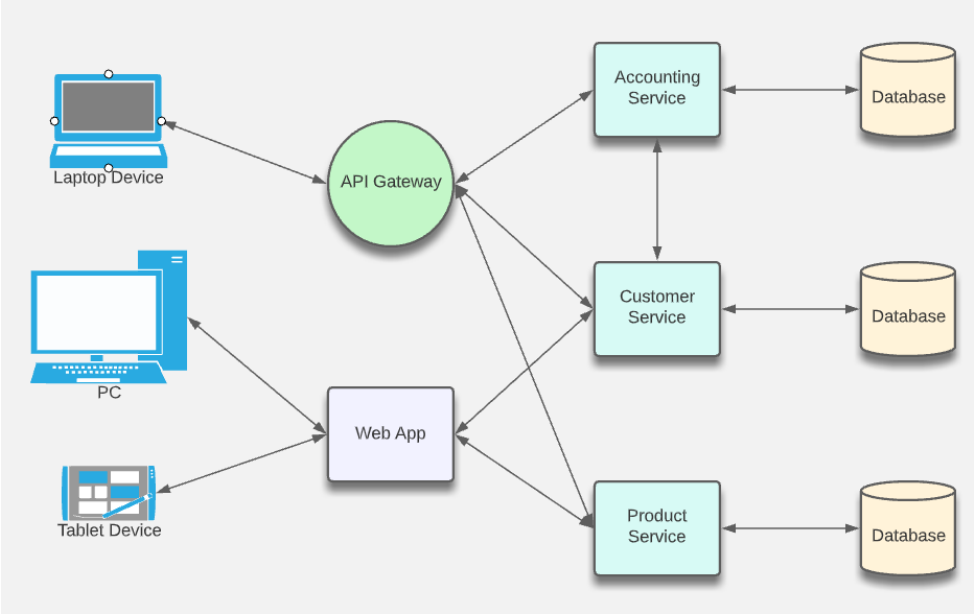
Microservices Architecture

Benefits

- Enables work in parallel
- Promotes organization according to business domain
- Advantages from isolation
- Flexible in terms of deployment and scale
- Flexible in terms of technology
- Allows multiple agile teams to maximize autonomy in effort and technology



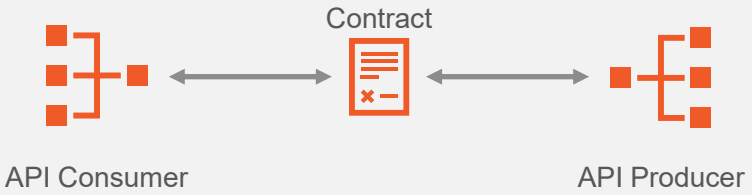
Microservices Architecture



Implications

- Requires a different way of thinking
- Complexity moves to the integration layer
- Organization needs to be able to support re-org according to business domain (instead of technology domain)
- With an increased reliance on the network, you may encounter latency and failures at the network layer
- Transactions must be handled differently (across service boundaries)

Where Microservices & APIs Overlap



APIs:

- Model the exchange of information between a consumer & producer
- Support business integration between two internal systems (“in process”) or between an internal system & an external partner (“out of process”)
- Built around technology specifications for interaction & type definition
- Enable integration over the network/Internet

80

Technology Options for Building APIs

When deciding on a technology to use for building an API, today's landscape offers multiple options:

- ✓ Implementation using a serverless platform like AWS Lambda
- ✓ Exposing via Robotic Process Automation (RPA)
- ✓ Exposing as a set of endpoints using the Mulesoft platform (or similar)
- ✓ Implementation using a microservices-based approach & architecture

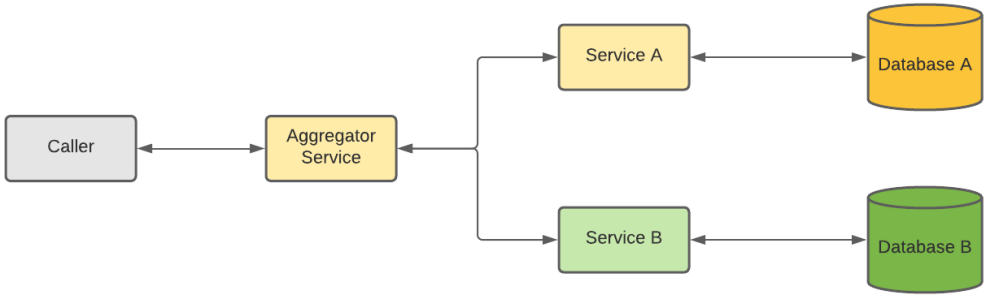
Microservice Design Patterns

- Aggregator
- API Gateway
- Chain of Responsibility
- Asynchronous Messaging
- Circuit Breaker
- Anti-Corruption Layer
- Strangler Application
- Others as well (https://microservices.io/patterns/index.html)

82

Microservice Design Patterns - Aggregator

- Akin to a web page invoking multiple microservices and displaying results of all on single page
- In the pattern, one microservice manages calls to others and aggregates results for return to caller
- Can include update as well as retrieval ops



Microservice Design Patterns - Aggregator

Benefits:

- Helps you practice DRY (Don’t Repeat Yourself)
- Supports logic at time of aggregation for additional processing

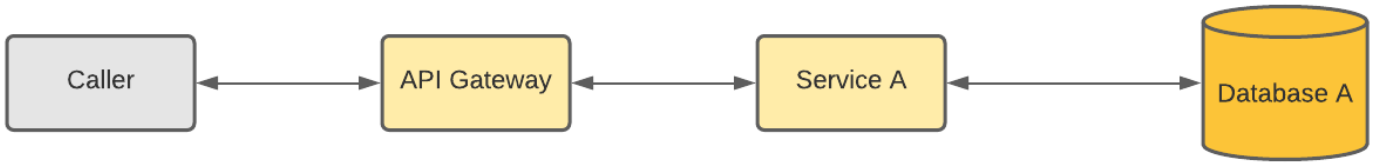
Potential “gotchas”:

- Performance if downstream microservices not called asynchronously

84

Microservice Design Patterns – API Gateway

- Specific type of infrastructure that helps manage the boundary
- Provides an intermediary for routing calls to a downstream microservice
- Provides a protection and translation layer (if calling protocol different from microservice)
- Can be combined with other patterns as well
- Considered a best practice from a security perspective as well



Microservice Design Patterns – API Gateway

Benefits:

- Usually, built in throttling (to protect against DDoS)
- Can support translation between caller and downstream microservice for protocol, message format, etc.
- Provides a centralized point of entry for AuthN/AuthZ
- Can be combined with load balancing to enable scalability and resiliency
- Centralized logging

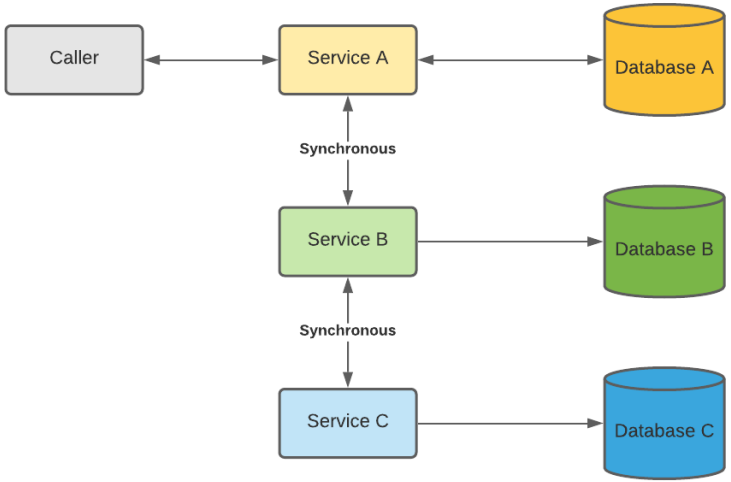
Potential “gotchas”:

- Can increase cost
- Requires additional configuration & management but enables additional capabilities as well

86

Microservice Design Patterns – Chain of Responsibility

- Represents a chained set of microservice calls to complete a workflow action
- Output of 1 microservice is input to the next
- Uses synchronous calls for routing through chain



Microservice Design Patterns – Chain of Responsibility

Benefits:

- Supports composition of microservices in a workflow that must happen in sequence
- Synchronous communication typically less complex

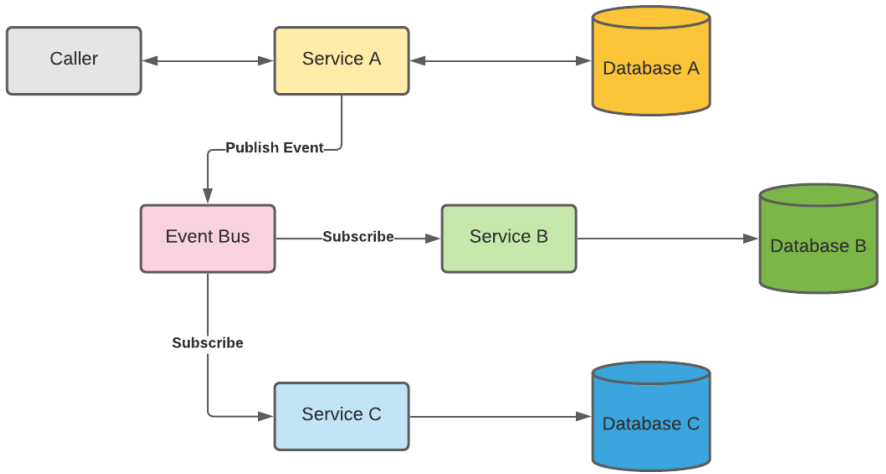
Potential “gotchas”:

- Increased response time – response time becomes the sum of each microservice’s response time in the chain

88

Microservice Design Patterns – Asynchronous Messaging

- Supports asynchronous interaction between independent microservices
- Messages published to a topic by a producer
- Topic subscribed to by one or more consumers



Microservice Design Patterns – Asynchronous Messaging

Benefits:

- Promotes looser coupling in cases where synchronous calls are not required
- Allows multiple services the option of being notified (vs. point-to-point)
- If given consumer is down, producer can continue to send messages and they won't be lost

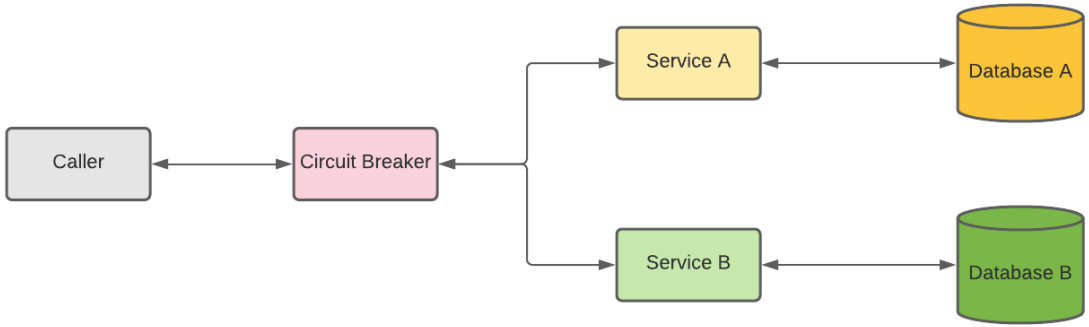
Potential "gotchas":

- Additional complexity
- Can be harder to trace an action end-to-end (but there are ways to handle)
- Not a good fit if specific timing and sequencing required

90

Microservice Design Patterns – Circuit Breaker

- Prevents unnecessary calls to microservices if down
- Circuit breaker monitors failures of downstream microservices
- If number of failures crosses a threshold, circuit breaker prevents any new calls temporarily
- After defined time period, sends a smaller set of requests and ramps back up if successful



Microservice Design Patterns – Circuit Breaker

Benefits:

- Helps to optimize network traffic by preventing calls that are going to fail
- Can help prevent “noise” and network overload

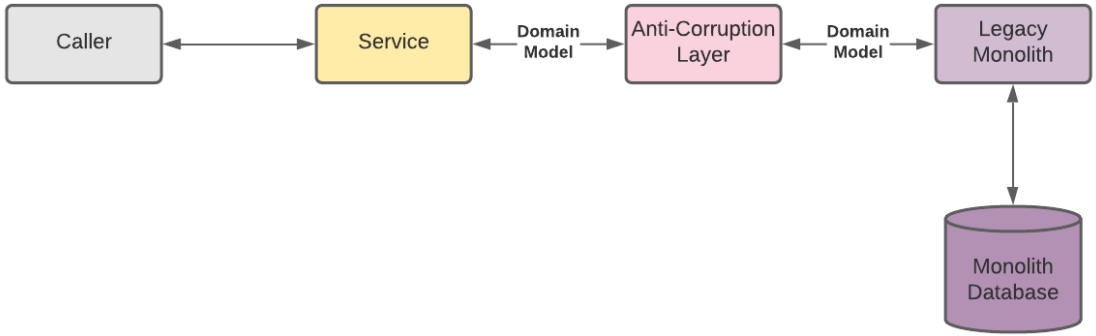
Potential “gotchas”:

- Process supported needs to be able to absorb limited downtime
- If not managed correctly, can result in poor user experience

92

Microservice Design Patterns – Anti-Corruption Layer

- Prevents corruption of new microservice domain model(s) with legacy monolith domain model(s)
- Provides a proxy/translation layer to help keep models “clean”



Microservice Design Patterns – Anti-Corruption Layer

Benefits:

- Prevents crossing of boundaries and “leaking” of sub-optimal logic
- Helps to keep this insulation transparent between communicating parties

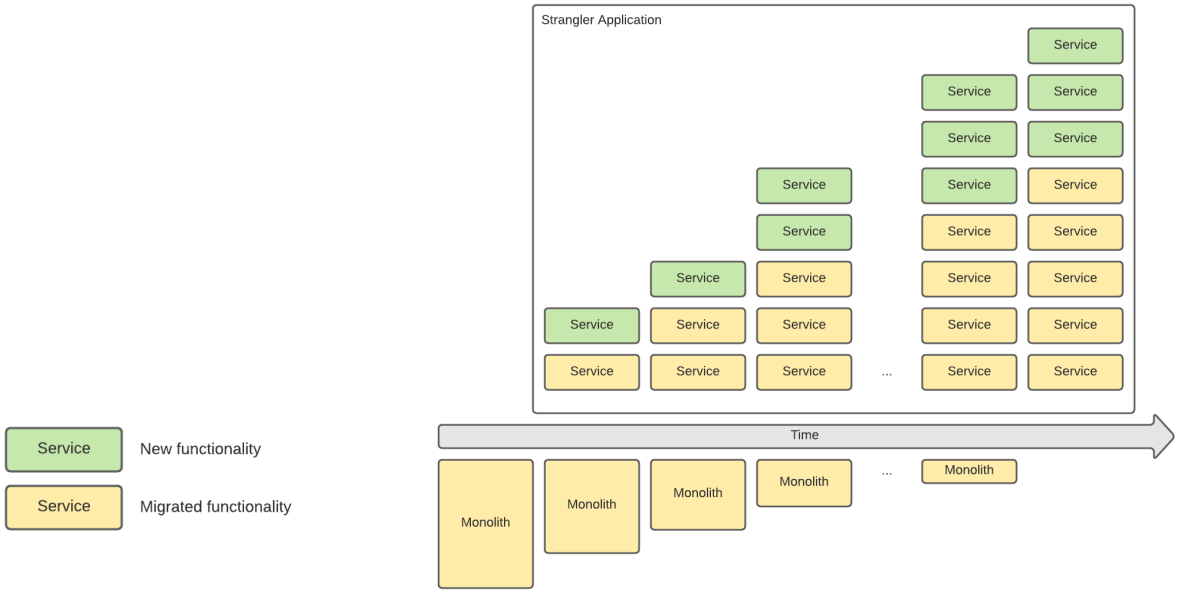
Potential “gotchas”:

- Additional complexity
- Additional testing required to validate the additional complexity

94

Microservice Design Patterns – Strangler Application

- Supports breaking up a monolith into microservices over time
- Can accommodate existing as well as new functionality



Microservice Design Patterns – Strangler Application

Benefits:

- Allows a gradual vs. “big bang” breakup
- Especially well-suited for large monoliths that are taking active traffic

Potential “gotchas”:

- Requires business to be able to absorb gradual
- Adds complexity – coordinating new functionality, migrated functionality, and proper integration between them
- While transition in progress, will have to maintain two paths

96

Group Discussion:

Microservices Architecture

As a group, discuss the pros & cons of this architectural style. Specifically:

- What are some advantages to using this style? Is there a specific application profile that would cause you to prefer this style?
- What are some disadvantages to using this style? Is there a specific application profile that would you to avoid this style?

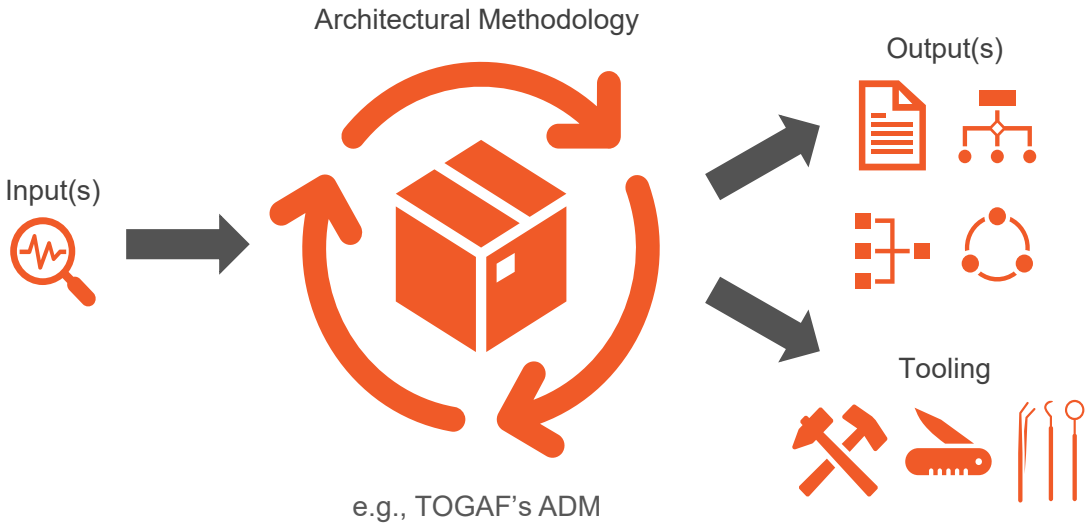
It’s the Most Wonderful Time of the Year

https://github.com/KernelGamut32/working-with-design-patterns-public/blob/main/use-cases/Toyshop.pdf

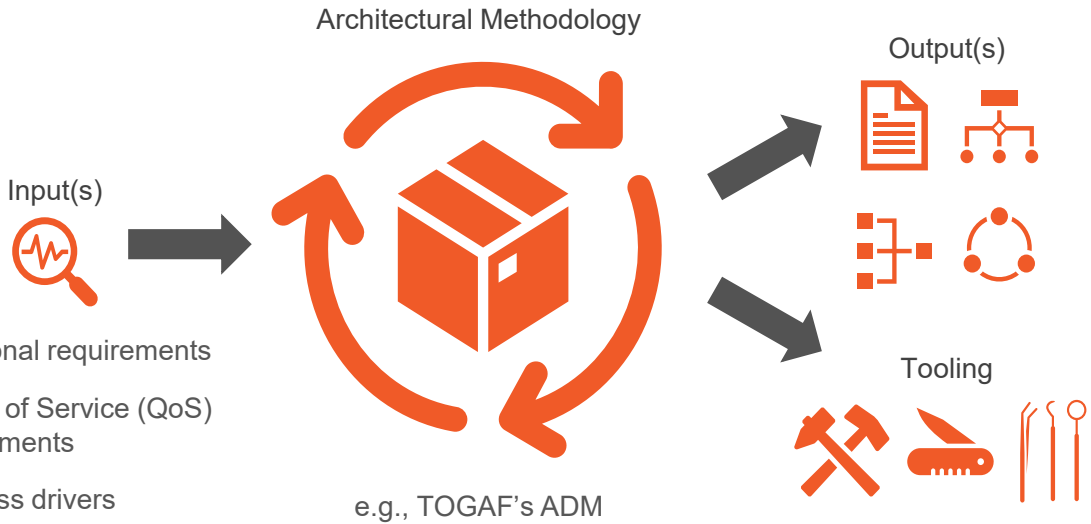
Architecture Design in Practice

Outputs & Tooling

Architectural Artifacts

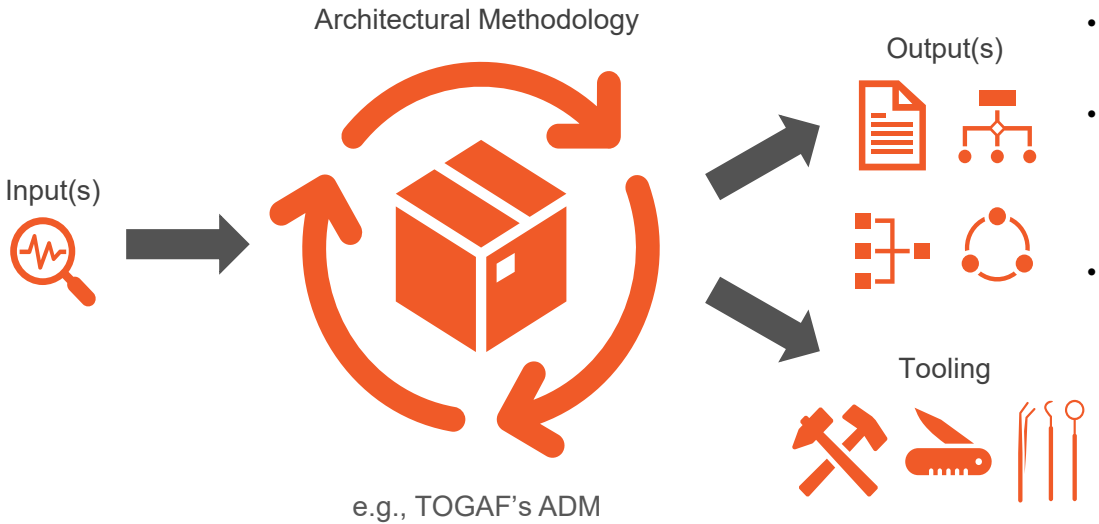


Architectural Artifacts



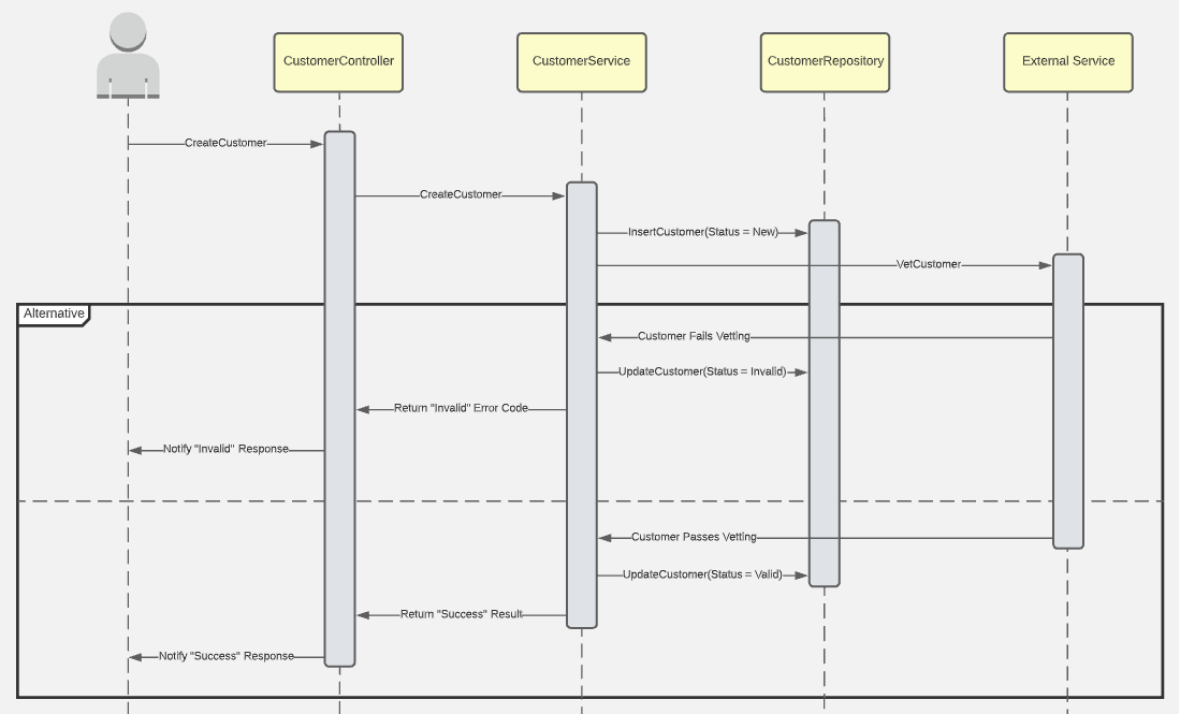
- Functional requirements
- Quality of Service (QoS) requirements
- Business drivers
- Technical drivers
- Organizational drivers

Architectural Artifacts

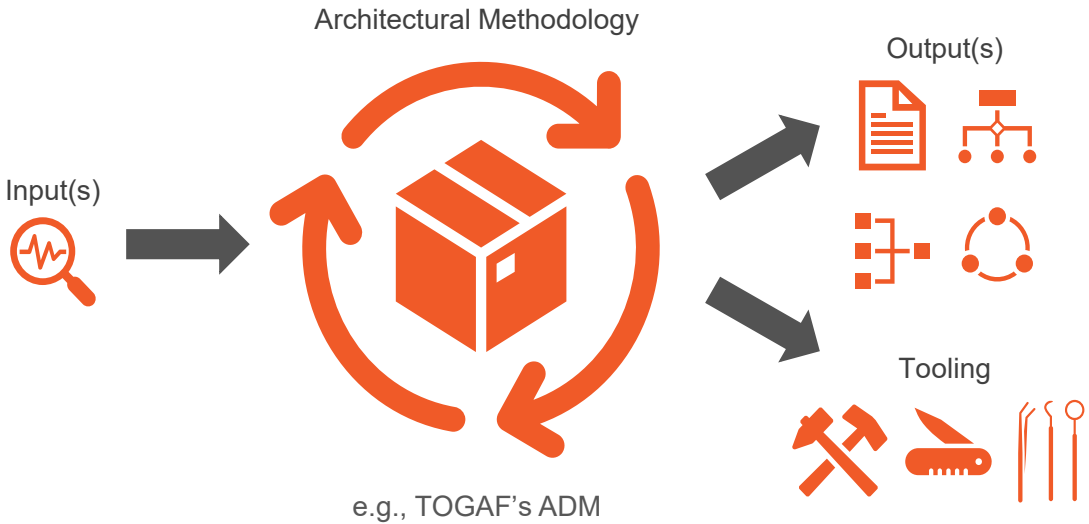


- Documentation of standards & best practices
- Diagrams (e.g., sequence diagrams, logical/physical architecture diagrams, ERDs)
- Processes that enable consistent application of the capability of architecture

Sample Sequence Diagram

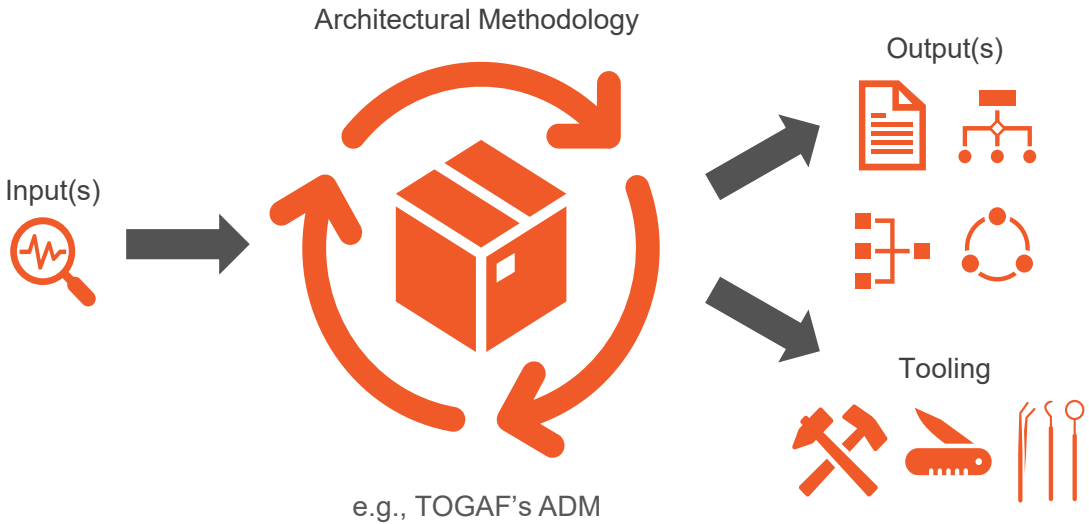


Architectural Artifacts



- SDKs, frameworks, libraries, etc. that can be integrated into new projects & designs
- Helps promote consistency & speed
- Making the right thing the "easy" thing

Architectural Artifacts



Couple of key points to keep in mind:

- Output & tooling may vary depending on architectural scope (enterprise, solution, domain), and all scopes should be considered
- Outputs & tooling must be easy to locate (search capabilities, known & maintained repositories, etc.); the architectural methodology should assist with cataloguing

Use Case – Outputs & Tooling

As a squad, you would like to "sell" to the business a modernization effort to migrate a **monolithic, mainframe-based application** used to manage a **critical sales & distribution process** within the company to a **microservices** architecture. Included in the target goals would be the transition of the workflow from the data center to **Cloud and Cloud native technologies** (where possible).

Currently, the application leverages a mainframe-based relational data store to store and serve all data within the large and tightly-coupled workflow. There are several legacy technologies in play and the platform upon which the application is deployed is **several versions behind latest**. Key reports that the business needs for critical decisioning exist but **cannot be delivered in real-time**. They are requested in batch and sent as a follow-up, sometimes 1 or 2 days later.

Also, the business would like to be able to leverage the different dimensions of data housed within this system for **Machine Learning** purposes but, at present, the data is difficult to extract and utilize. The current mainframe system supports **external customer access through a Web portal**, **internal access for account & order management** (**including remote sales associates utilizing a mobile device**), and a handful of API endpoints that are used by **external partners** to enable **authorized third-party sales**.

In this scenario:

- What types of architecture artifacts would you anticipate producing? How might this project use them? How might future projects use them?
- What types of reusable tooling might arise out of the execution of this project effort?

Let’s Play Sequence!

https://github.com/KernelGamut32/working-with-design-patterns-public/tree/main/use-cases/uml

Quality of Service Requirements

Quality attributes as Architecture Requirements

What are some practical ways to architect for these key QoS requirements?

Five fundamental quality attributes

Scalability
Ability to address increasing and fluctuating volumes with acceptable response time

Availability/Reliability
Ability to ensure high availability and recover from failures

Flexibility
Ability to add, modify and remove features and capabilities

Operability
Ability to support smooth operations and easy maintenance

Security
Ability to secure the enterprise assets from internal and external threats

Quality attributes as Architecture Requirements

- Includes a couple of options – scaling up & scaling out (scaling out most common type with microservices architectures)
- Statelessness is a useful paradigm
- Scaling rules can be resource-based or schedule-based
- Goal should be mechanisms that automate scaling out & scaling in
- Scaling in also important for cost management

Five fundamental quality attributes

Scalability
Ability to address increasing and fluctuating volumes with acceptable response time

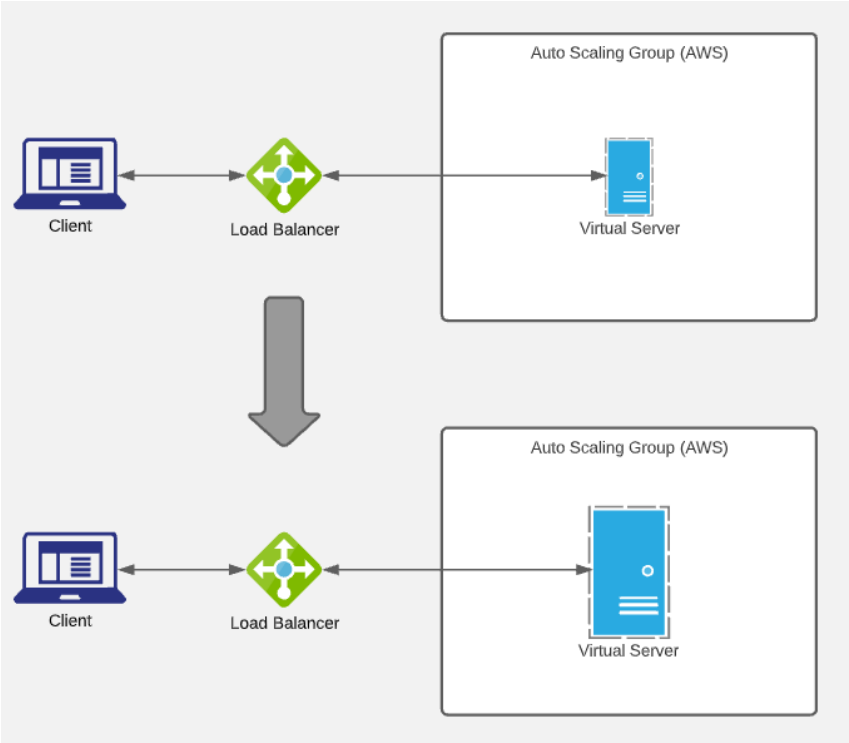
Availability/Reliability
Ability to ensure high availability and recover from failures

Flexibility
Ability to add, modify and remove features and capabilities

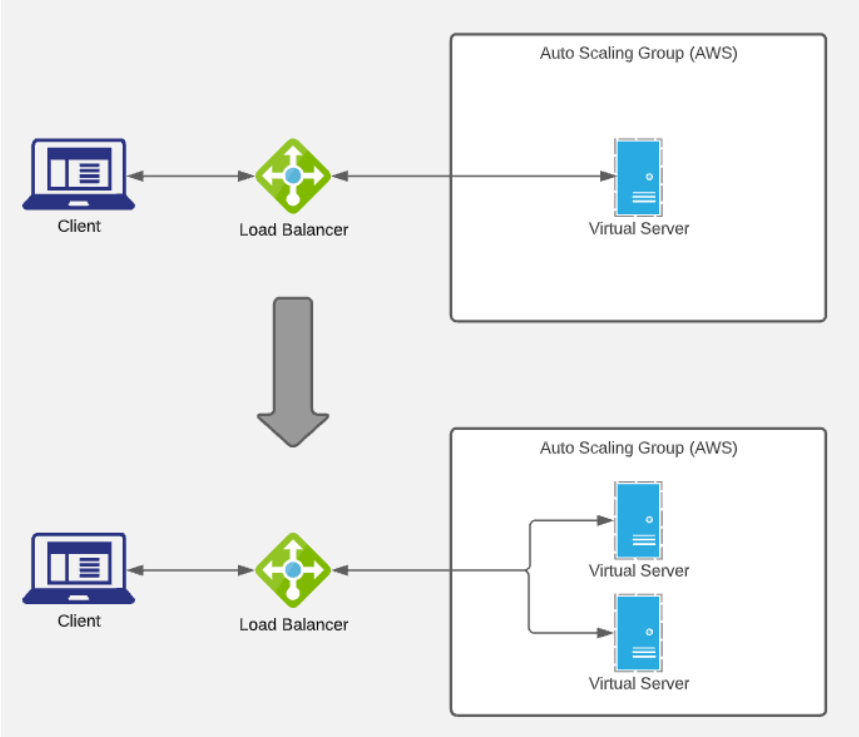
Operability
Ability to support smooth operations and easy maintenance

Security
Ability to secure the enterprise assets from internal and external threats

Vertical Scaling (Scaling “Up”)



Horizontal Scaling (Scaling “Out”)



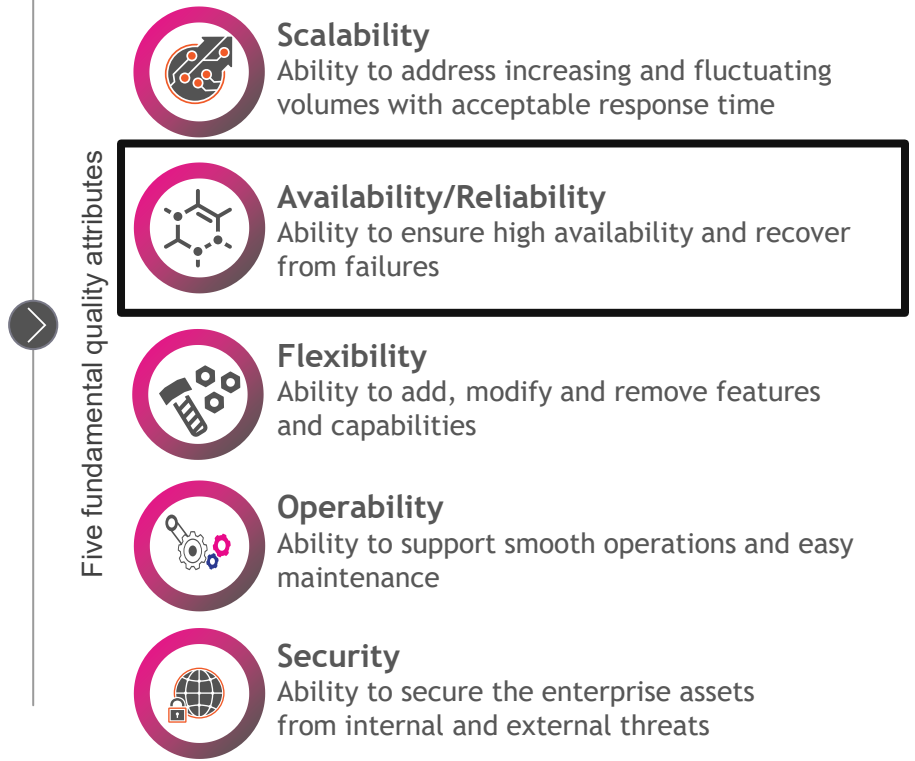
Moment of Reflection

What are some pros & cons of each type of scaling (scaling “up” vs. scaling “out”)?

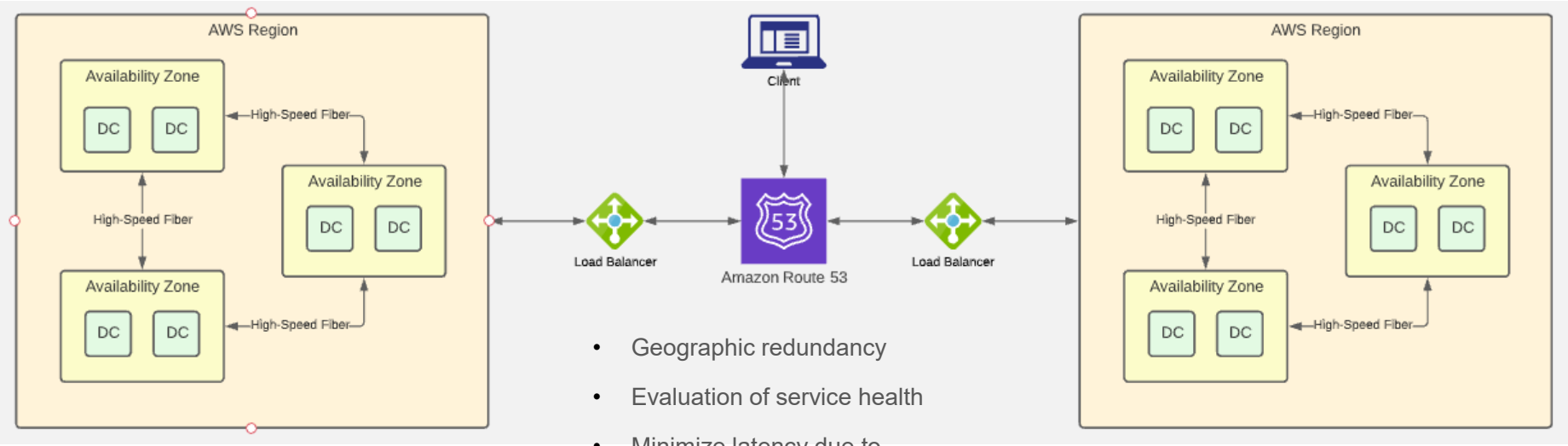
Type your answer in chat or come off mute & share

Quality attributes as Architecture Requirements

- Need a clear understanding of minimum SLAs
- What are the requirements for geographic redundancy? Across the country or across the globe?
- Management of latency and distance data has to travel
- If transactional, is active-passive sufficient or do you require active-active?
- Recovery Time Object (RTO) & Recovery Point Objective (RPO) are important data points
- Cost optimization



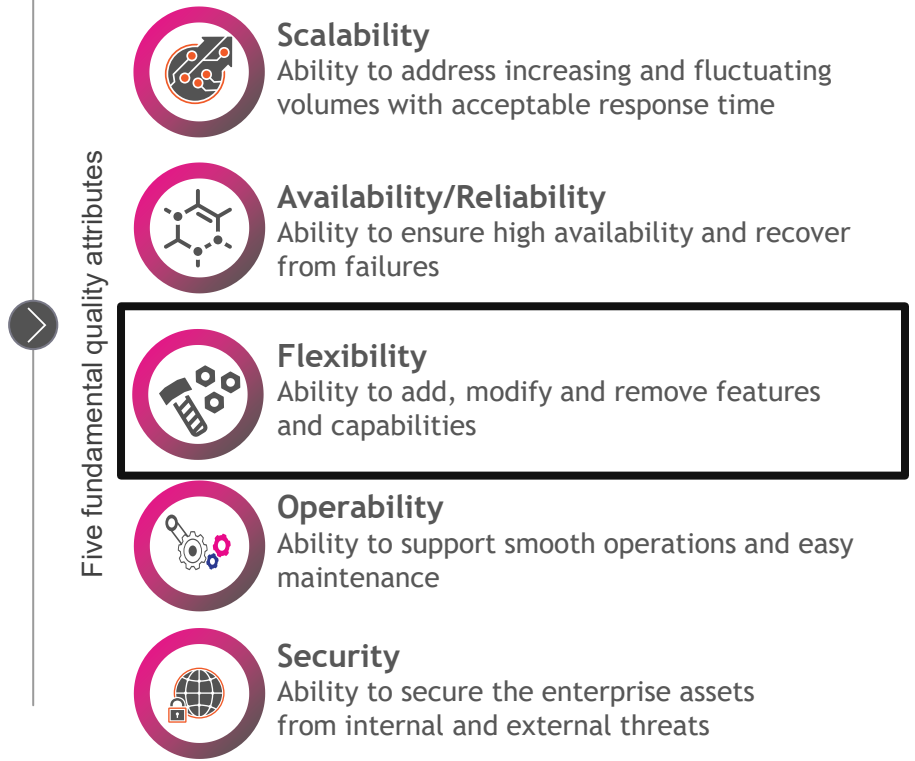
Availability/Reliability



- Geographic redundancy
- Evaluation of service health
- Minimize latency due to distance

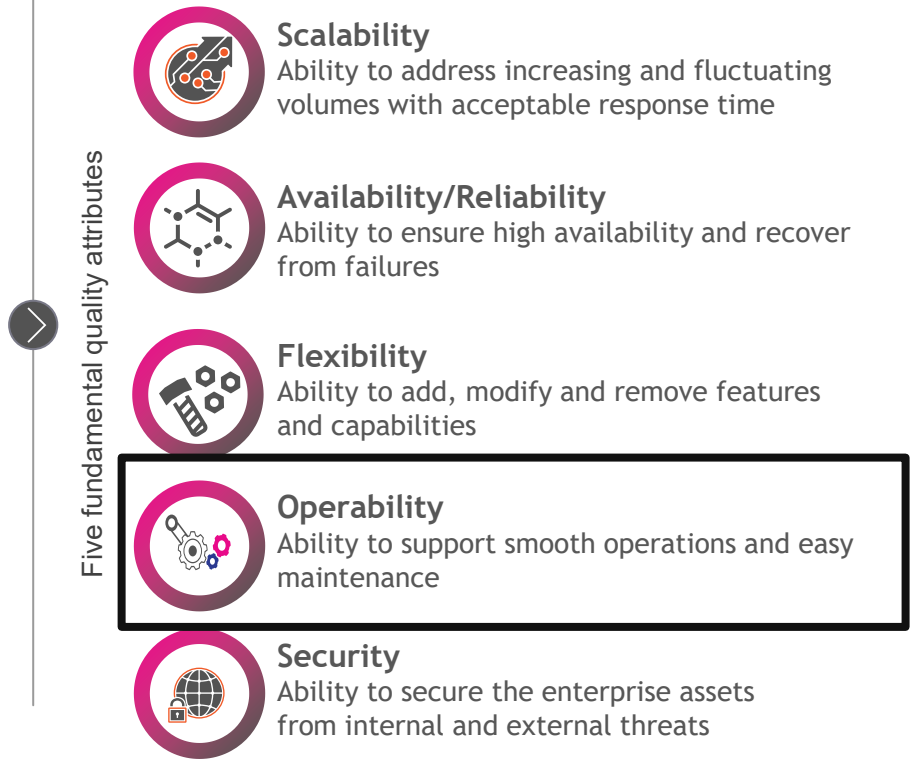
Quality attributes as Architecture Requirements

- Ensuring that the architecture promotes loose coupling
- Leveraging design patterns that facilitate adding & removing features with minimized impact
- Maintaining a proper versioning strategy, especially if requirement to run multiple versions side-by-side
- Design to abstractions rather than concretions
- Utilize techniques like Dependency Injection
- Make maintainability a key goal of the design

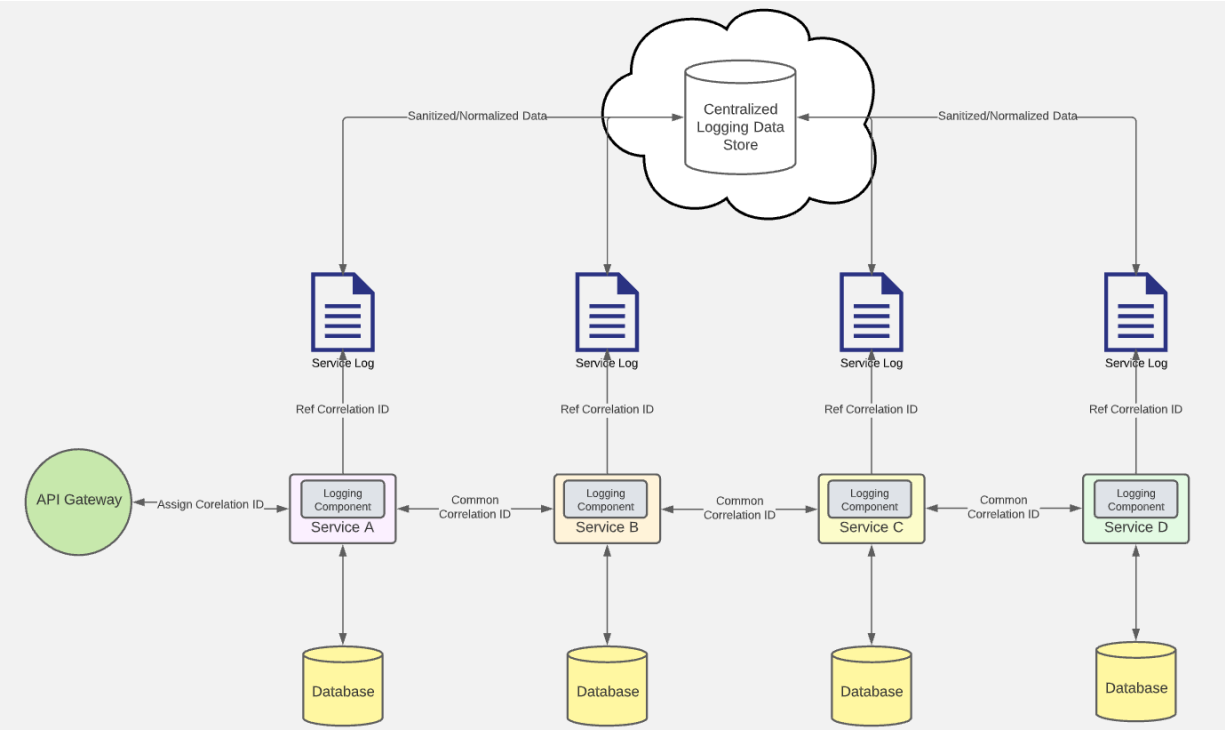


Quality attributes as Architecture Requirements

- Use known observability patterns for logging & tracing transactions through the system
- Likely able to find multiple frameworks to enable – don't reinvent the wheel
- Ensure sanitization of relevant log data to prevent leakage of sensitive information
- Especially in a microservices architecture, end-to-end traceability becomes critical
- Correlation IDs can be transmitted across a series of microservice requests (e.g., in headers) to tie together
- Ask yourself "What would I need to know to support this system in production if I'm getting paged at 3 a.m.?"

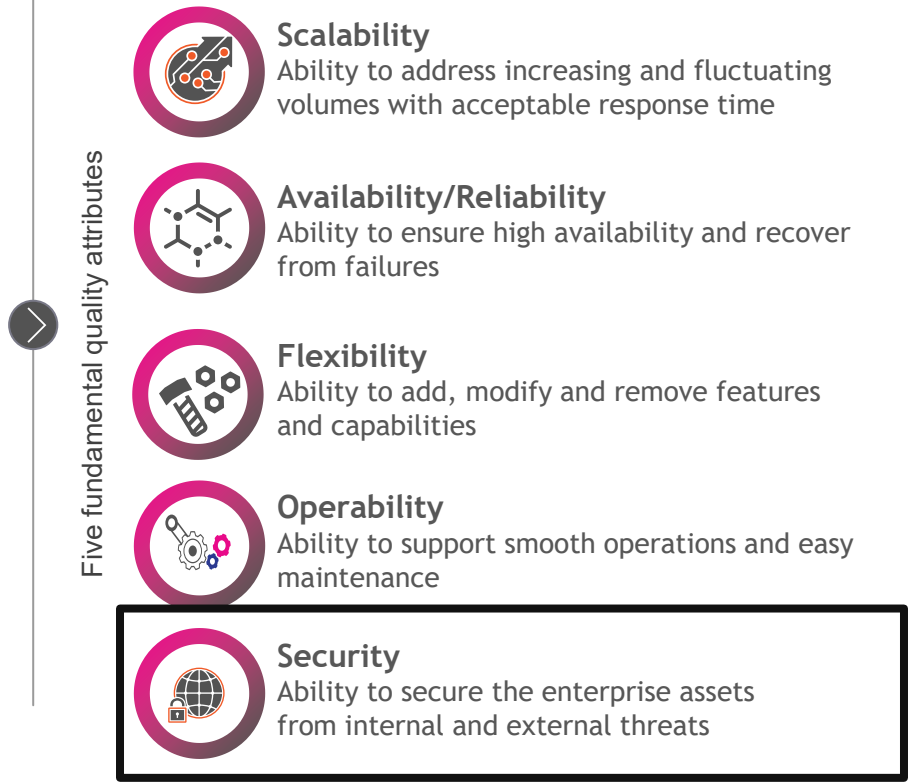


Operability

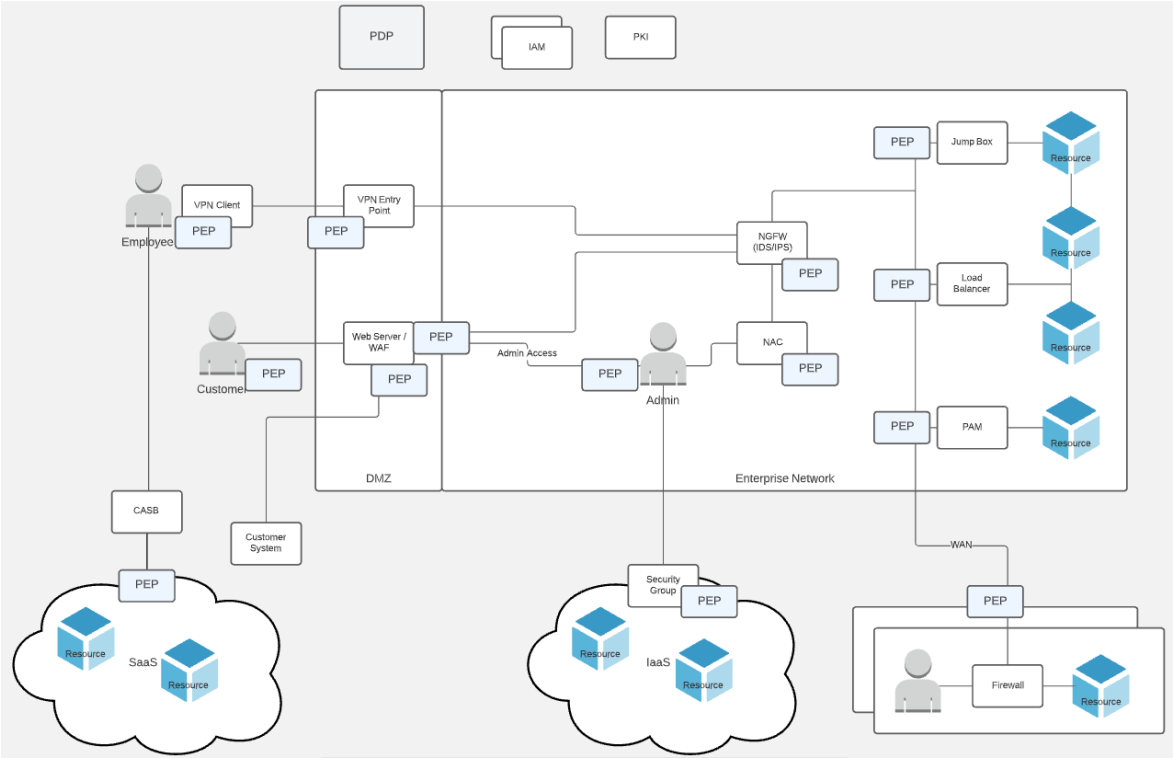


Quality attributes as Architecture Requirements

- Practice key tenets of security architecture:
  - Defense in depth
  - Least privilege
  - Secure by default
- Utilize principles of Zero Trust Architecture to help you avoid “castle & moat”, minimizing zones of implicit trust within the system & its components
- Ideally, the architecture of the system will be built around an established DevSecOps framework which seeks to “shift security left”



Security – Zero Trust Architecture



Use Case – QoS Requirements

As a squad, you would like to “sell” to the business a modernization effort to migrate a **monolithic, mainframe-based application** used to manage a **critical sales & distribution process** within the company to a **microservices** architecture. Included in the target goals would be the transition of the workflow from the data center to **Cloud and Cloud native technologies** (where possible).

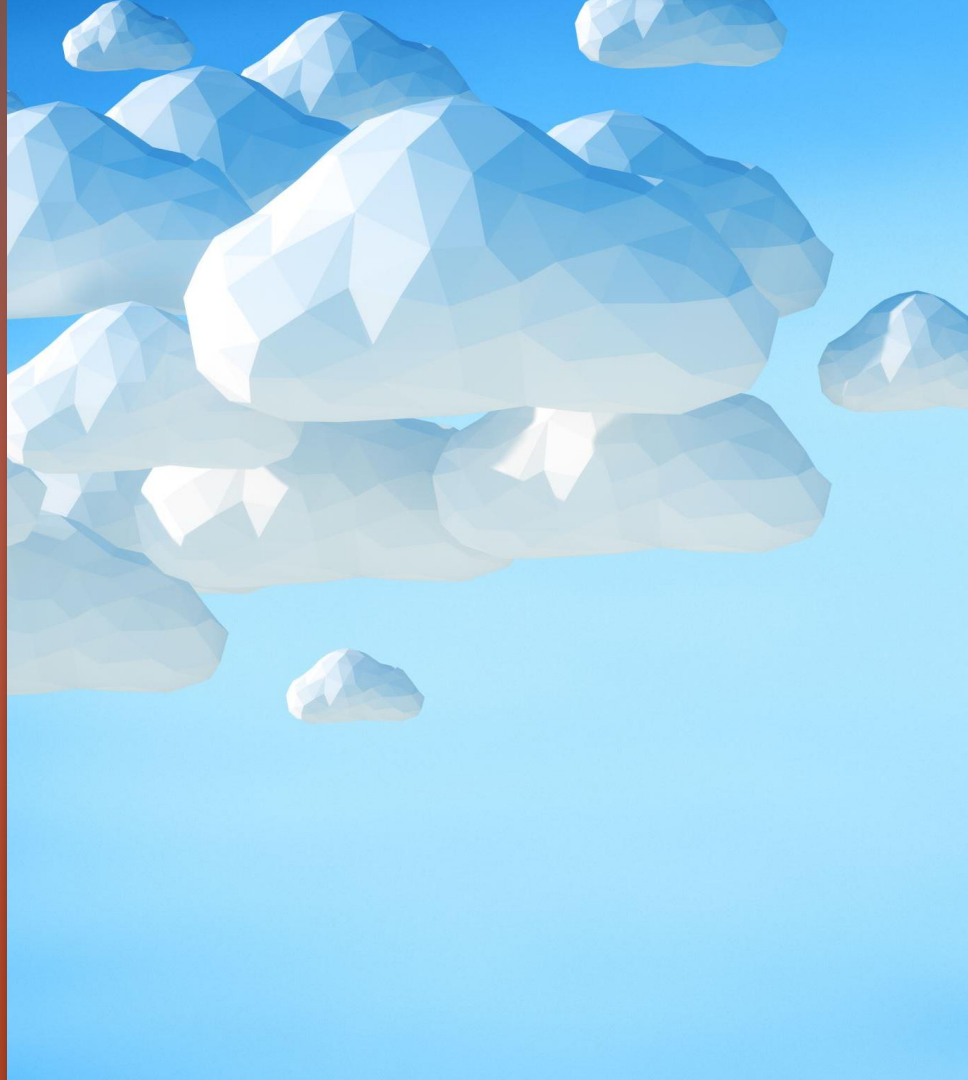
Currently, the application leverages a mainframe-based relational data store to store and serve all data within the large and tightly-coupled workflow. There are several legacy technologies in play and the platform upon which the application is deployed is **several versions behind latest**. Key reports that the business needs for critical decisioning exist but **cannot be delivered in real-time**. They are requested in batch and sent as a follow-up, sometimes 1 or 2 days later.

Also, the business would like to be able to leverage the different dimensions of data housed within this system for **Machine Learning** purposes but, at present, the data is difficult to extract and utilize. The current mainframe system supports **external customer access through a Web portal**, **internal access for account & order management** (**including remote sales associates utilizing a mobile device**), and a handful of API endpoints that are used by **external partners** to enable **authorized third-party sales**.

In this scenario:

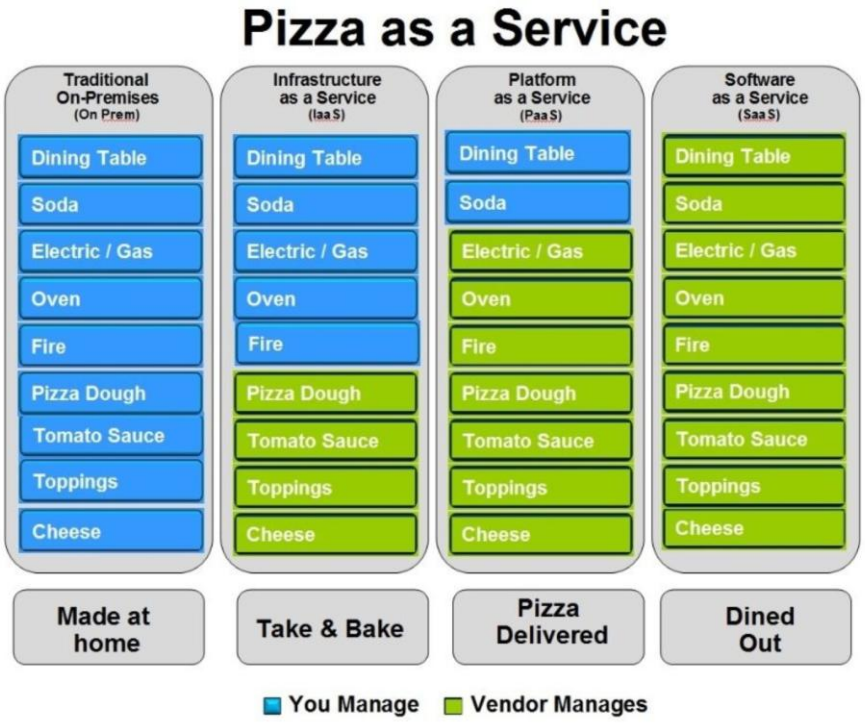
- How would you prioritize the 5 key QoS requirements for this use case?
- What are some practical ways you could architect for each?
- How might you measure adherence (or discover deficiency)?

Architecting for the Cloud

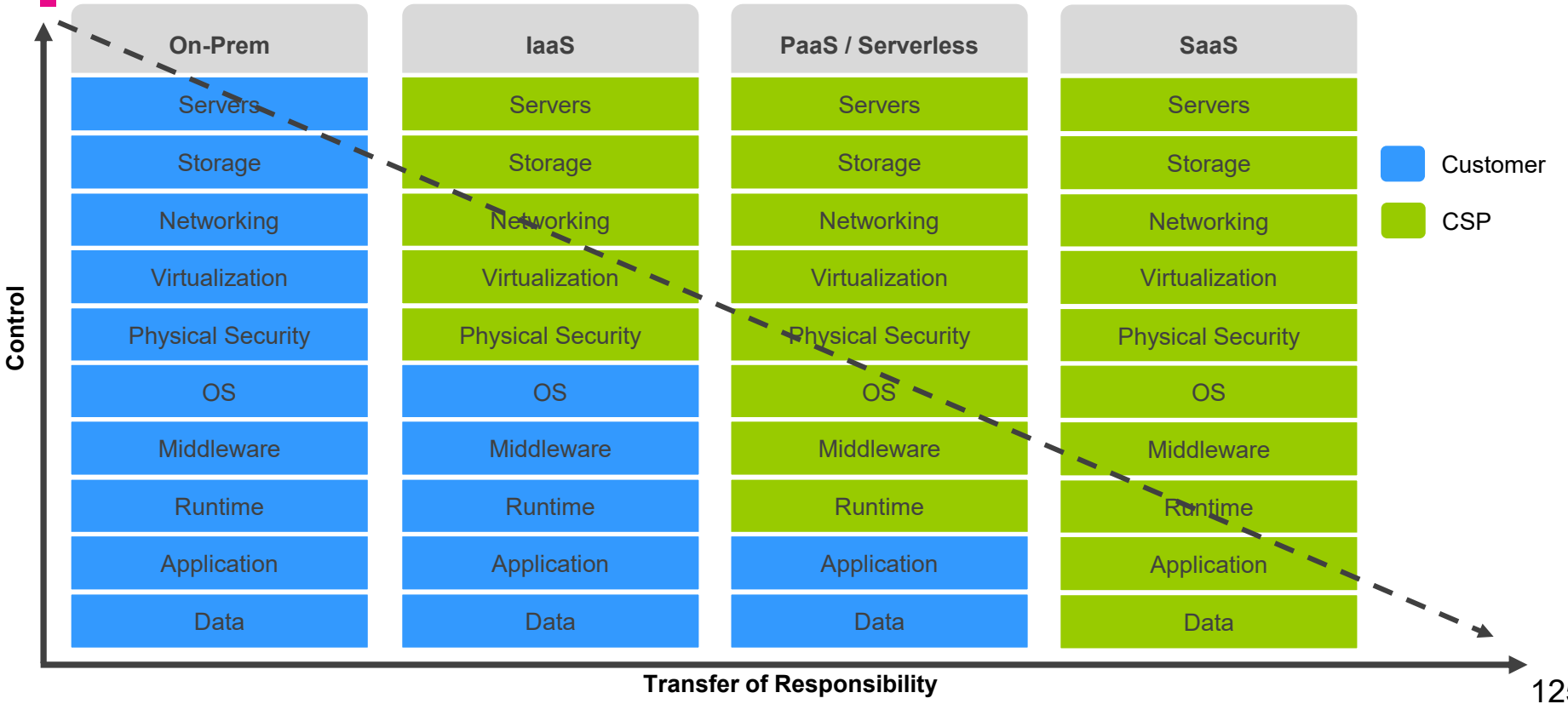


Pizza-as-a-Service

From a LinkedIn post by Albert Barron from IBM (https://www.linkedin.com/pulse/20140730172610-9679881-pizza-as-a-service/)

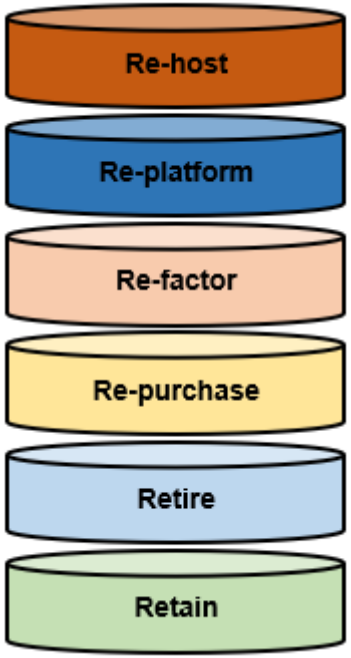


Side-by-Side Comparison



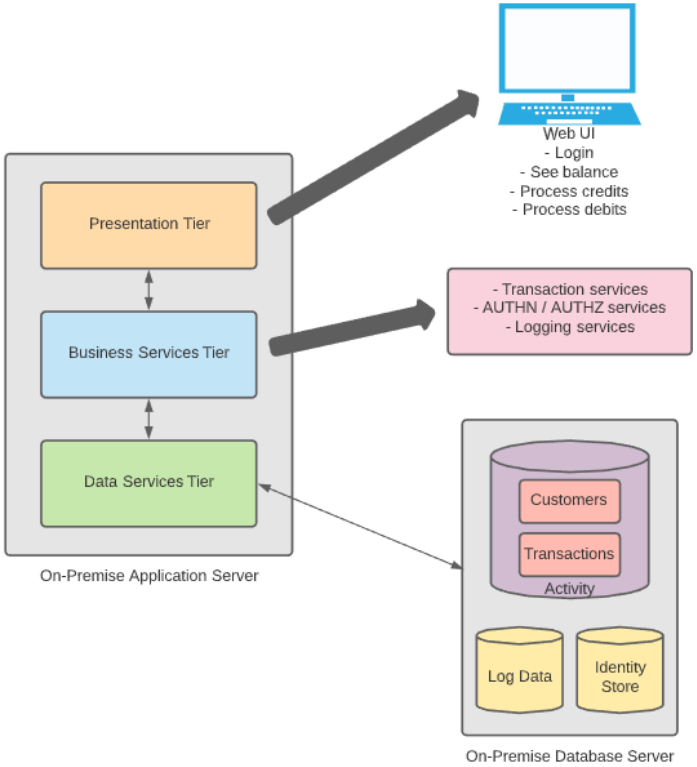
The 6 R's – Application Migration Strategies

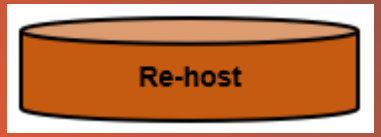
When moving from on-premise to the Cloud (or hybrid), the architect decides whether to migrate, modernize, or do some combination of the two



Source: https://aws.amazon.com/blogs/enterprise-strategy/6-strategies-for-migrating-applications-to-the-cloud/

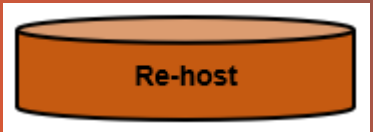
Current State Architecture

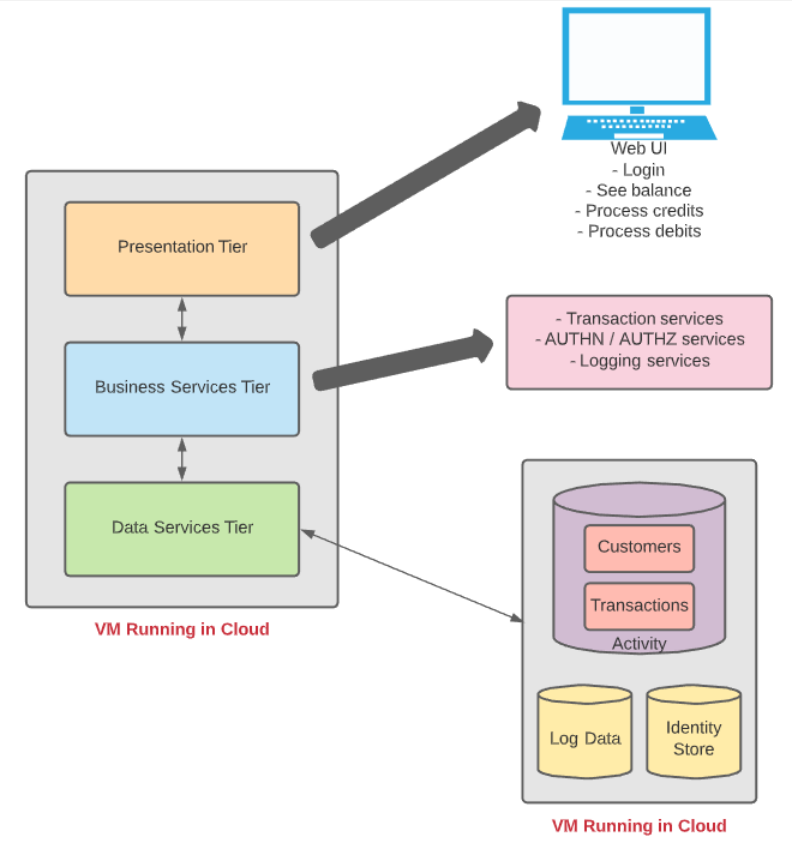




- AKA “lift & shift”
- For all intents and purposes, involves recreating the on-premise infrastructure in the Cloud
- Sometimes used to expedite retirement of a data center

- Can be a mechanism to quickly migrate workloads and see immediate cost savings, even without Cloud optimizations
- There are third-party tools available to help automate the migration
- Once the application is in the Cloud, it can be easier to apply Cloud optimizations vs. trying to migrate and optimize at the same time

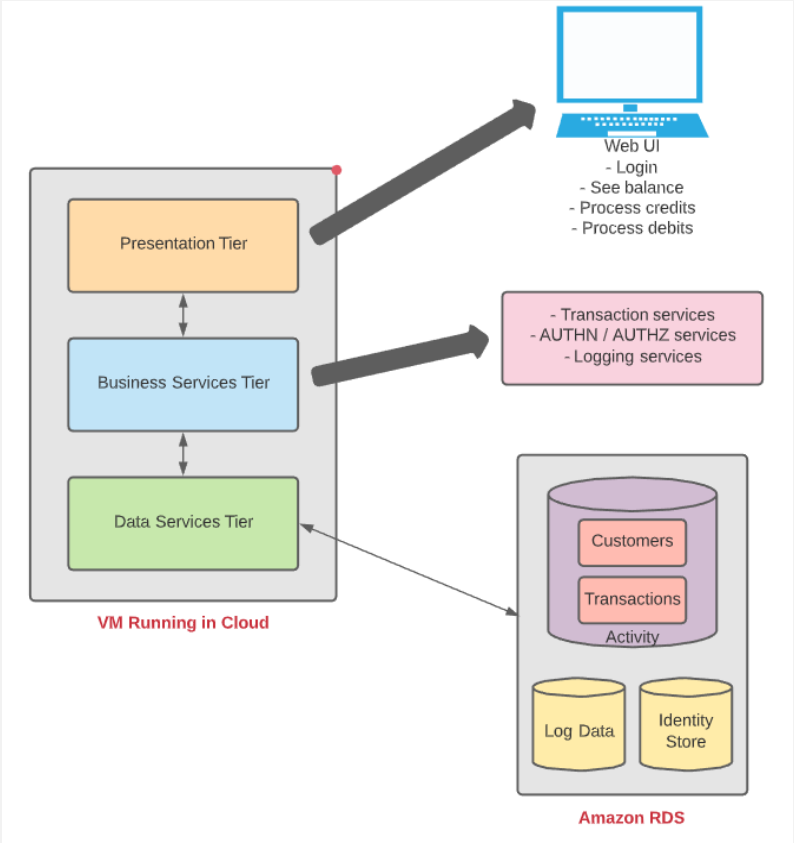

Re-host

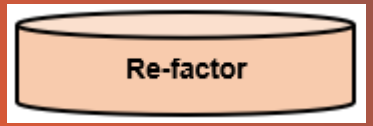

Web UI
- Login
- See balance
- Process credits
- Process debits
Presentation Tier
Business Services Tier
Data Services Tier
VM Running in Cloud
- Transaction services
- AUTHN / AUTHZ services
- Logging services
Customers
Transactions
Activity
Log Data
Identity Store
VM Running in Cloud

- AKA “lift, tinker & shift” or “lift & fiddle”
- Core architecture of the application will not change
- Involves recreating most of the on-premise infrastructure in the Cloud with a few Cloud optimizations

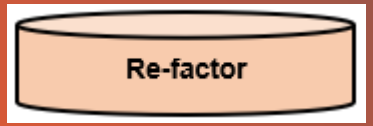
- Those optimizations will involve a move to one or more Cloud native services for a specific, tangible business benefit
- A common example is moving to a managed database (e.g., Relational Database Service, or RDS, in AWS)
- Enables migration speed while providing cost savings or benefit in a targeted portion of the application’s architecture


Re-platform

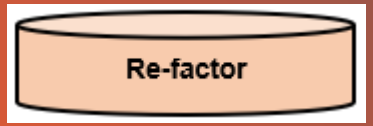

Web UI
- Login
- See balance
- Process credits
- Process debits
Presentation Tier
Business Services Tier
Data Services Tier
VM Running in Cloud
- Transaction services
- AUTHN / AUTHZ services
- Logging services
Customers
Transactions
Activity
Log Data
Identity Store
Amazon RDS



- Involves rearchitecting the application to maximize utilization of Cloud native optimizations
- Likely the most expensive and most complex of the available options
- The application profile needs to fit, and business value must be identified commensurate with the cost required to execute



- Good option if the application can benefit from features, scalability, or performance offered by the Cloud
- Examples include rearchitecting a monolith to microservices running in the Cloud or moving an application to serverless technologies for scale (like the use case we've been discussing)



?

To Be Determined

- Good fit for applications that are candidates for moving from on-premise licensing (for installed products) to a SaaS model
- Often applications that have been installed to manage a specific type of business capability
- Examples can include Customer Relationship Management (CRM), Enterprise Resource Planning (ERP), HR or e-mail (among others)

- In some cases, an enterprise may have a “healthy” percentage of legacy applications that are no longer being used or maintained (especially for larger organizations)
- When completing the Application Portfolio Assessment and associated interviews with application owners, look for opportunities to recommend retire of the unused legacy apps
- This type of application can represent a “quick win”

- The associated savings can potentially be advantageously factored into the business case for the Cloud modernization effort
- Finally, retiring unused apps reduces attack surface area from a security perspective

- Applications that must remain in place but that cannot be migrated to the Cloud without major refactor
- This type of application profile may prevent the ability to completely move out of the data center (at least in the interim)
- From a cost perspective, it can be difficult for an enterprise to take on both the operating expense of Cloud while continuing to carry the capital expense of a data center and on-premise infrastructure
- The hybrid model can be a good solution to support where needed

Reference Architectures

AWS Well-Architected Framework

https://aws.amazon.com/architecture/well-architected

AWS Reference Architectures

https://aws.amazon.com/architecture/

Architecting for the Cloud

Hosting approach & cloud services to be utilized

Security requirements & effective method(s) to implement

Observability requirements & operational management strategy

Data management, classification, & protection

Quality of Service (QoS) requirements & the “-ilities”

Use Case – Architecting for the Cloud

As a squad, you would like to “sell” to the business a modernization effort to migrate a **monolithic, mainframe-based application** used to manage a **critical sales & distribution process** within the company to a **microservices** architecture. Included in the target goals would be the transition of the workflow from the data center to **Cloud and Cloud native technologies** (where possible).

Currently, the application leverages a mainframe-based relational data store to store and serve all data within the large and tightly-coupled workflow. There are several legacy technologies in play and the platform upon which the application is deployed is **several versions behind latest**. Key reports that the business needs for critical decisioning exist but **cannot be delivered in real-time**. They are requested in batch and sent as a follow-up, sometimes 1 or 2 days later.

Also, the business would like to be able to leverage the different dimensions of data housed within this system for **Machine Learning** purposes but, at present, the data is difficult to extract and utilize. The current mainframe system supports **external customer access through a Web portal**, **internal access for account & order management** (**including remote sales associates utilizing a mobile device**), and a handful of API endpoints that are used by **external partners** to enable **authorized third-party sales**.

In this scenario, assume the intent is to migrate the application out of the data center and to the Cloud:

- What types of Cloud service (IaaS, PaaS, FaaS, SaaS) would you recommend leveraging and for which components?

Design Patterns

Design Patterns

- As previously discussed, design patterns are proven solutions to a specific technical problem
- Focused primarily on the level of the source code
- Different classes of problem/solution that can be used (and reused) as building blocks

Factory Pattern

- Creational pattern used to abstract creation logic from client
- Rather than create directly, code uses the factory to generate new instances
- Provides way to vary what gets created (and how) based on business logic

Abstract Factory Pattern

- Creational pattern used to abstract creation logic from client
- Sometimes called factory of factories

LAB 07:

Abstract Factory Pattern

https://github.com/KernelGamut32/working-with-design-patterns-public/tree/main/labs/lab07

Singleton Pattern

- Creational pattern used to manage class instance
- Seeks to create a single instance and provide a point of global access to that single instance
- Supports early and lazy instantiation patterns
- Likely for a specific type of use case – e.g., configuration detail

LAB 08:

Singleton Pattern

https://github.com/KernelGamut32/working-with-design-patterns-public/tree/main/labs/lab08

Prototype Pattern

- Creational pattern used to create instances by cloning rather than creating new
- Can be useful when intent is to minimize the number of instances or when cost of creating a new instance is high/complex

Builder Pattern

- Creational pattern used to build out an object over the course of multiple steps
- Each of the steps can be represented by a different object (if required)
- Enables clear codification of the construction approach

LAB 09:

Prototype/Builder Patterns

https://github.com/KernelGamut32/working-with-design-patterns-public/tree/main/labs/lab09

Adapter Pattern

- Structural pattern used to wrap an interface
- Enables internal structure of object to remain as-is with layering in of a new class that can adapt specific client needs
- Allows reuse of existing functionality (even if not directly compatible)

Bridge Pattern

- Structural pattern used to decouple abstraction from implementation
- Supports extensibility and hides implementation details from code that references
- Allows abstraction and implementation to vary independently

Composite Pattern

- Structural pattern used to enable reference and access to a group of objects using a top-level object reference
- Can be used to represent a tree or object hierarchy
- Useful for cases like org charts or graphs of related objects

LAB 10:

Adapter, Bridge, Composite Patterns

https://github.com/KernelGamut32/working-with-design-patterns-public/tree/main/labs/lab10

Flyweight Pattern

- Structural pattern used to reduce the number of objects created, thereby reducing memory footprint and helping to improve performance
- Seeks to reuse objects where possible, only creating new if existing not found
- Supports shareable state through exposure of same object (if available) to multiple clients

LAB 11:

Flyweight Pattern

https://github.com/KernelGamut32/working-with-design-patterns-public/tree/main/labs/lab11

Decorator Pattern

- Structural pattern used to add new functionality to an existing object without altering its underlying, internal structure
- Clear example of OCP (Open for Extension, Closed for Modification)
- Uses composition instead of inheritance

Facade Pattern

- Structural pattern used to present a simplified interface that can be leveraged to facilitate access to a more complex, underlying structure
- Hides complexities, making downstream component/structure easier to use
- Promotes loose coupling through simplified abstraction

LAB 12:

Decorator & Facade Patterns

https://github.com/KernelGamut32/working-with-design-patterns-public/tree/main/labs/lab12

Chain of Responsibility Pattern

- Behavioral pattern used to process a given request through a chain of objects, any of which can handle the request
- Reduces coupling by giving multiple classes the opportunity to service the request
- Enables a sequence of options, processing through the sequence until an appropriate handler is found

Command Pattern

- Behavioral pattern used to encapsulate a handler for a specific type of command in a separate object
- Provides separation between object that issues the command and the agent fulfilling it
- Allows representation of a set of logic as a “thing” that can be passed around for encapsulated execution when needed

LAB 13:

Chain of Responsibility & Command Patterns

https://github.com/KernelGamut32/working-with-design-patterns-public/tree/main/labs/lab13

Observer Pattern

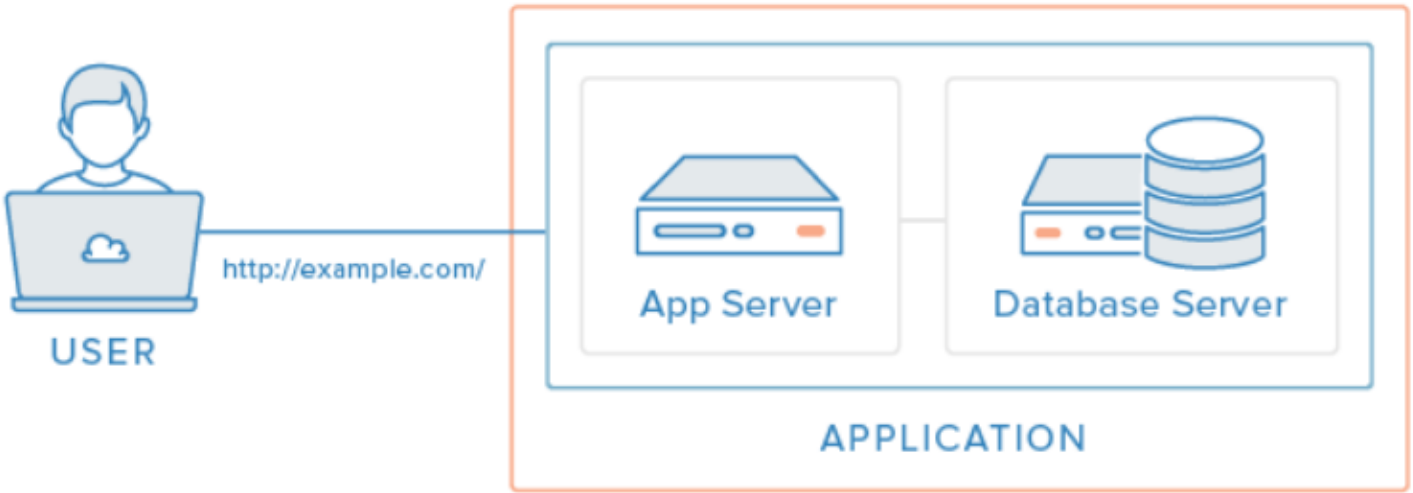
- Behavioral pattern used to manage a one-to-many relationship where multiple objects are watching and listening for activity in another
- Supports publish-subscribe approach – one publisher but can have multiple subscribers
- Event handling in programming languages is an example of the observer pattern

Strategy Pattern

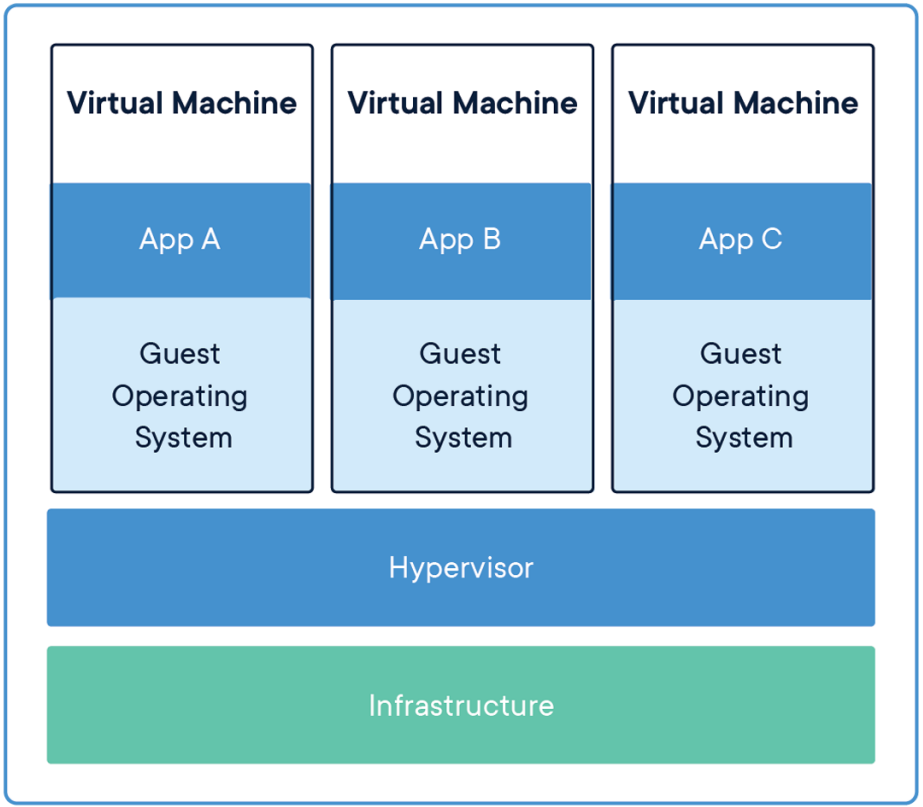
- Behavioral pattern used to handle changes to class behavior or algorithm at run time in response to executed logic
- Uses a controller or context to select algorithm to be executed based on assigned strategy

Kubernetes and Container Orchestration

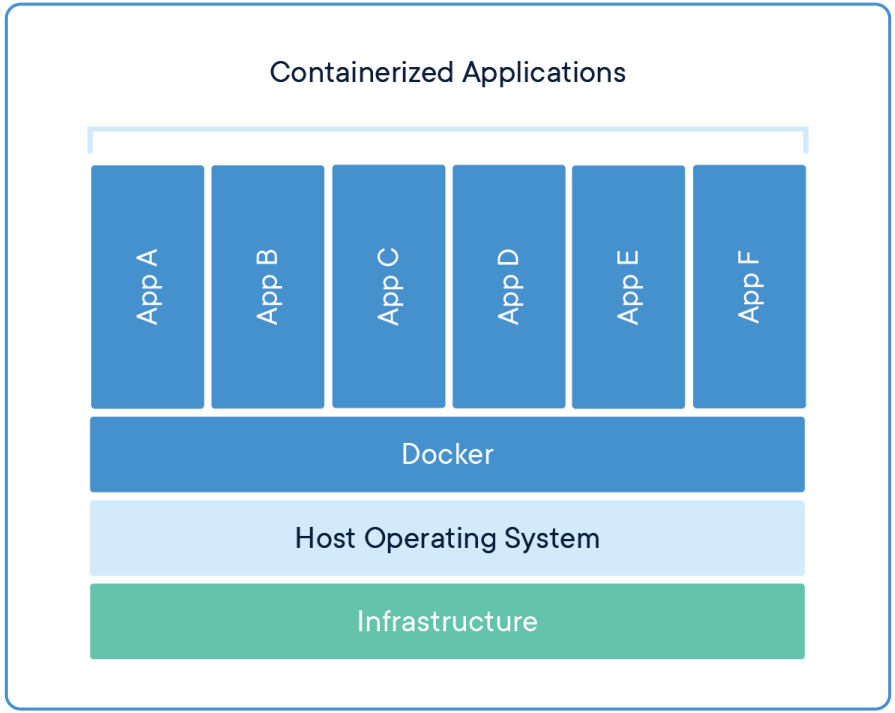
Evolution of Containers – Client/Server



Evolution of Containers – Virtual Machines



Evolution of Containers – Containers



What is Docker?

- Open-source containerization technology
- Enables deployment of self-contained & isolated application instances
- One of the foundational technologies for supporting microservices

What is Docker?

- Built around the concept of images & containers
- Also, supports composition of a set of containers to be deployed together
- For example, application code + network components + database components

What is Docker?

- Utilizes principles of “immutable infrastructure”
- Complete application environments torn down and recreated as needed
- Helps to minimize infrastructure “drift” and environment inconsistencies

The Dockerfile

- Tells Docker what to do in creating an image for your application
- The commands are all things you could do from the CLI
- Used by the docker “build” command
- Docker build uses this file and a “context” – a set of files at a specified location – to make your image

Dockerfile example

The following creates an image for building/running Java app in container
See https://github.com/KernelGamut32/dockerlab-repo-sample for sample

Dockerfile

```
# Grabs OpenJDK image upon which the new image will be based
FROM openjdk:17

# Creates a new target folder in image
RUN mkdir /usr/src/JavaDemoApp

# Copies current directory contents to newly created folder
COPY . /usr/src/JavaDemoApp

# Switches working directory in image to app folder
WORKDIR /usr/src/JavaDemoApp

# Compiles/builds Java app
RUN javac JavaDemo.java

# Executes new Java app
CMD ["java", "JavaDemo"]
```

Docker Images

- Represent templates defining an application environment
- New instances of the application can be created from the image
- These instances are called containers

Docker Images

- Images are defined via a Dockerfile definition
- Support layers for building up the environment in stages
- Fully defines the application, including all components required to support

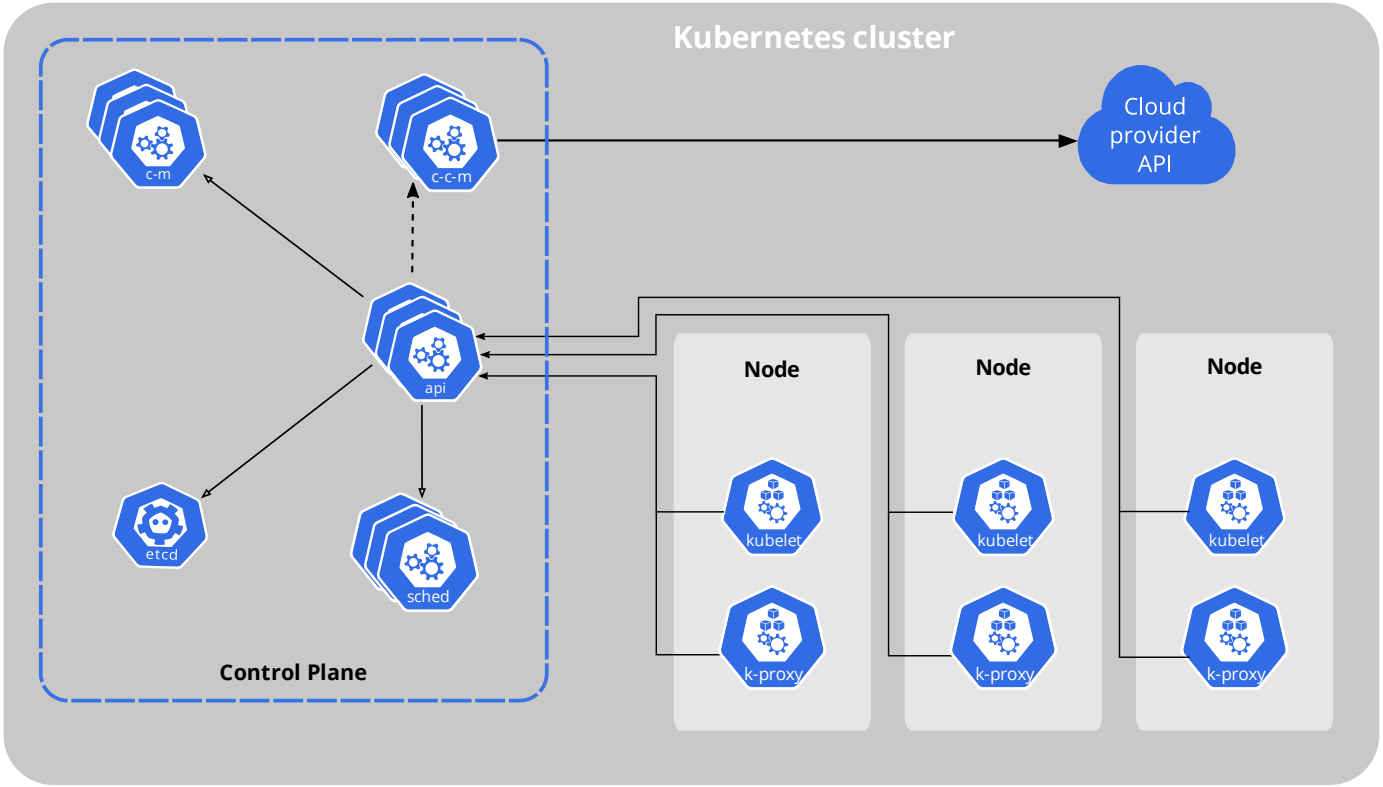
Kubernetes (k8s) Overview

- Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services

What is an Orchestrator and Why Do We Need It?

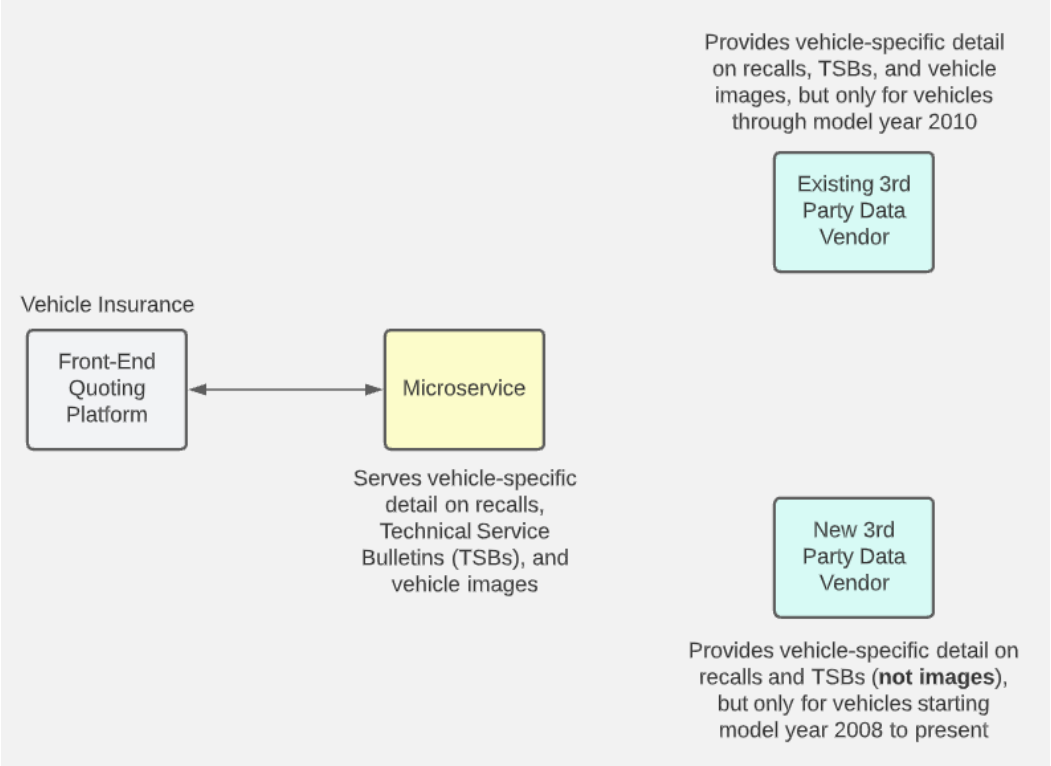
- What would it look like to manually control the containers needed for your application as your app scales or containers fail?
- “Orchestration” is the execution of a defined workflow
- We can use an orchestrator to start and stop containers automatically based on your set rules
- What does this open up for you?

Architecture of k8s System



Use Cases

An Exercise in Flexibility



How would you architect a flexible solution (i.e., maintainable, loosely-coupled, etc.) to this business problem?

Long Live the Mainframe

https://github.com/KernelGamut32/working-with-design-patterns-public/blob/main/use-cases/Mainframe%20Arch.pdf

Thank you!

If you have additional questions, please reach out to me at:
asanders@gamuttechnologysvcs.com

PLURALSIGHT