Welcome

Advanced Software Architecture

DevelopIntelligence
A PLURALSIGHT COMPANY

Hello

About me...

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

Prerequisites

This course assumes you:

- Have hands-on coding experience in a modern programming language (C#, C++, Java, Python, etc.)
- Have experience building moderate-sized software applications

Why study these subjects?

- 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

Agenda (Some of the Topics We'll Be Covering)

- Deep dive on key technologies, practices, and strategies that inform and enable software architecture
- Architecting for the Cloud
- Evolutionary architecture
- Microservices architecture
- Software architecture patterns
- Software design patterns

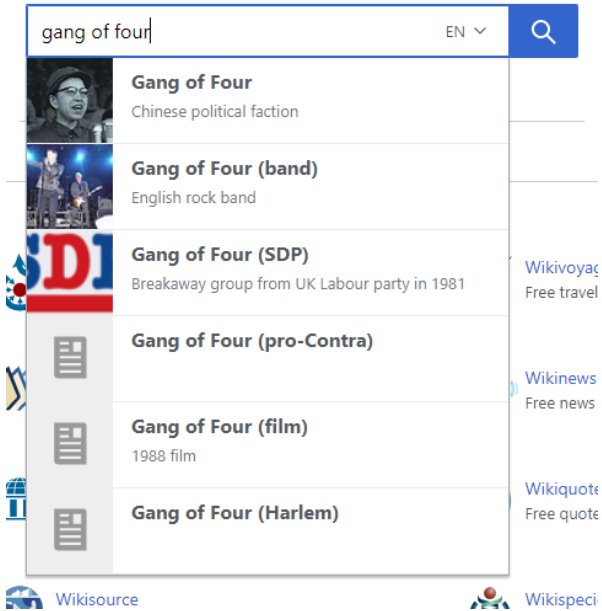
Origin of Design Patterns

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)

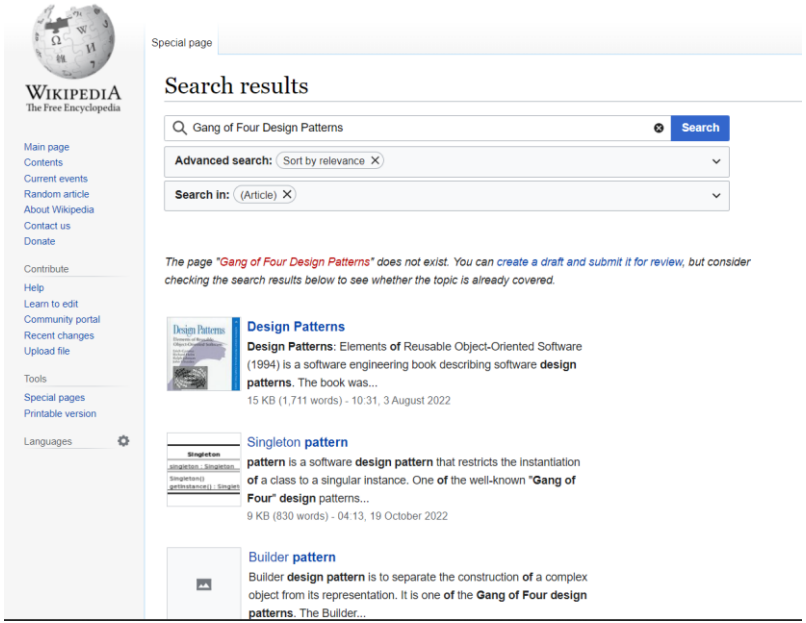
- Initial search for “Gang of Four” on www.wikipedia.org



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

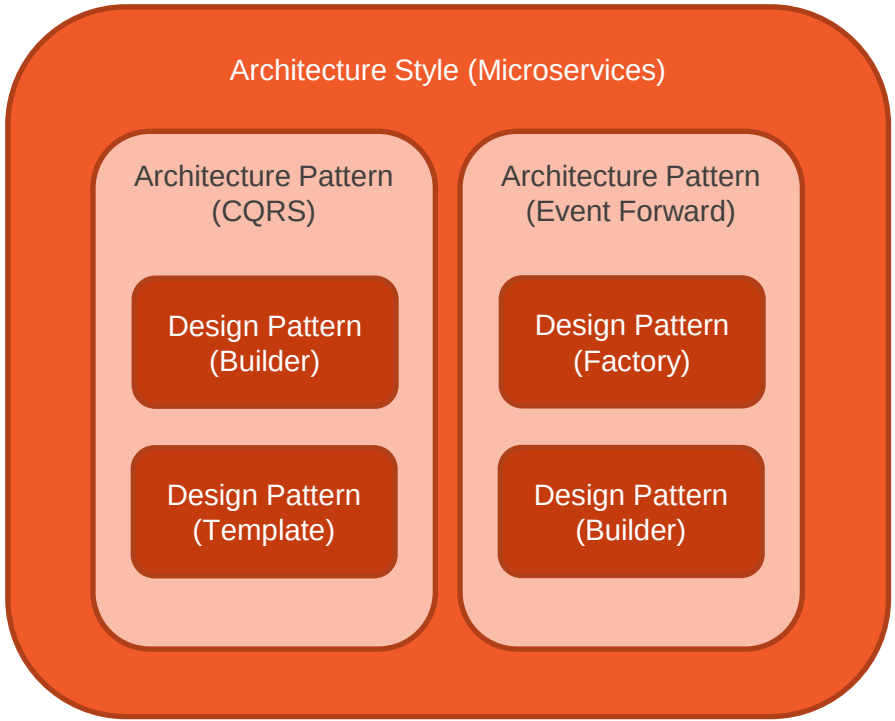
Architecture Style

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

Software Development Methodologies and Iterative Software Development

Common Approaches to Software Development

- Waterfall
- Iterative

Waterfall Software Development

A sequential development process that flows like a waterfall through all phases of a project (requirements, design, implementation, testing, and deployment for example), with each phase completely wrapping up before the next phase begins.

Key aspects:

- Majority of research done up front
- More accurate time estimates
- More predictable release date

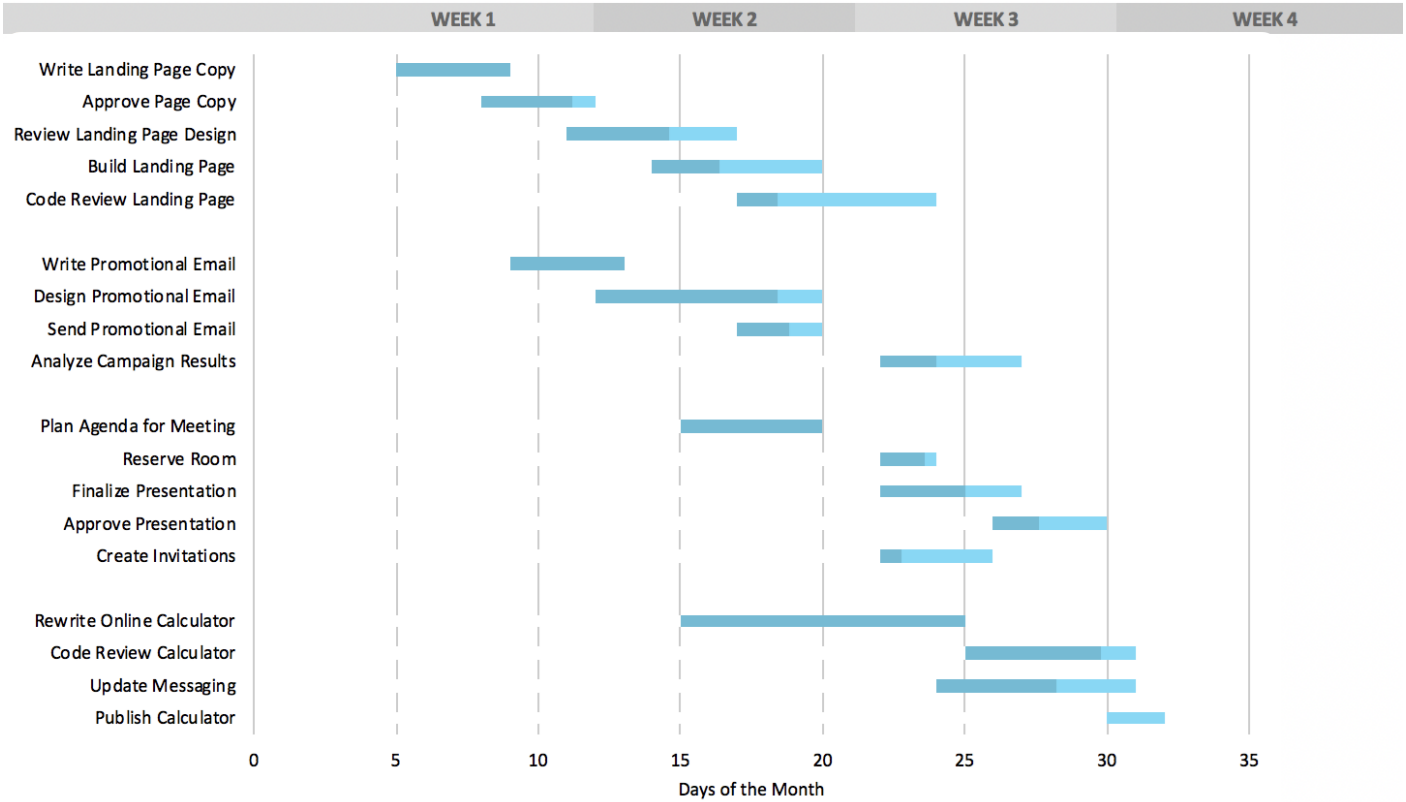
Cons:

- Process is brittle – can’t pivot easily in terms of changing requirements
- Long lead times – difficult to respond to rapid business evolution

Tools:

- Commonly use Gantt charts to track projects, subtasks and dependencies

Typical Gantt Chart



Agile Software Development

A group of software development methodologies based on iterative development.

Requirements and solutions evolve through collaboration between self-organizing cross-functional teams.

Key aspects:

- Incremental delivery
- Always ready to ship
- Continuous inspection of work product and process provides feedback for continuous improvement

We will explore Agile in more detail later in this class

Group Discussion:

Describe the software development or project management approaches you have worked with, and your opinions on the pros and cons of each

Notes:

- This shouldn’t be a long essay
- It’s more important that you communicate how the systems worked, in plain language – and what it was like to work in those systems. Talk about WHY you think something worked well, or didn’t
- Post responses in chat for review & discussion as a group

Iterative Software Development

Key elements:

- Incremental improvement to code and process
- Always ready to ship
- Iterative feature development
- Working with the Product Owner
- Pivoting mid-development

Incremental improvement to code and process

Principle: Using feedback obtained from regular examination of product and process lets you incrementally improve both

Key aspects:

- Work in short cycles (1-6 weeks) composed of overlapping phases: requirements, design, programming, testing.
- Build on what was built before and produce a working product at each stage. Gather customer feedback at the end of each iteration.
- Each iteration, evaluate the team’s performance. Change internal process as needed to improve quality of estimation and delivery.

Always ready to ship

Principle: after the first iteration, the product is always "ready to ship". That is, the product does something useful and is in a releasable state.

Key aspects:

- This is a risk mitigator – if the project is cancelled or greatly reduces in scope, there is still something of value to show for the work efforts
- Projects are less likely to be cancelled, as there is always a working product that shows the project’s value and the team’s competence

Key practices that support this principle:

- Do daily software builds
- Fix bugs right away
- Relentlessly use automated testing to mitigate regression

Iterative Feature Development

Principle: Instead of building all features and then releasing (waterfall), develop one feature, then the next, always keeping the application in a shippable state. Continually re-assess priority of feature development, including refinement of existing features, based on feedback.

Key aspects:

- Work closely with all concerned parties
- All involved need to understand and appreciate that development on project will be iterative
- Application reviewed at demo is still “work in progress”
- Focus is on gathering “real-time” feedback so course corrections can be made if required

Key practices that support this principle:

- Set expectations for all stakeholders
- Educate stakeholders on iterative development so they are effective contributors
- Be willing to change

Working with the Product Owner

Principle: working closely with the Product Owner enables the incremental production of shippable features that are valuable to the end user

Key aspects:

- Consult with the product owner before each iteration, agreeing on the scope and details of work for that iteration.
- During the iteration, the external customer or project manager cannot change the scope for that iteration, but the development team may change the scope by dropping features if the end date will not be met.

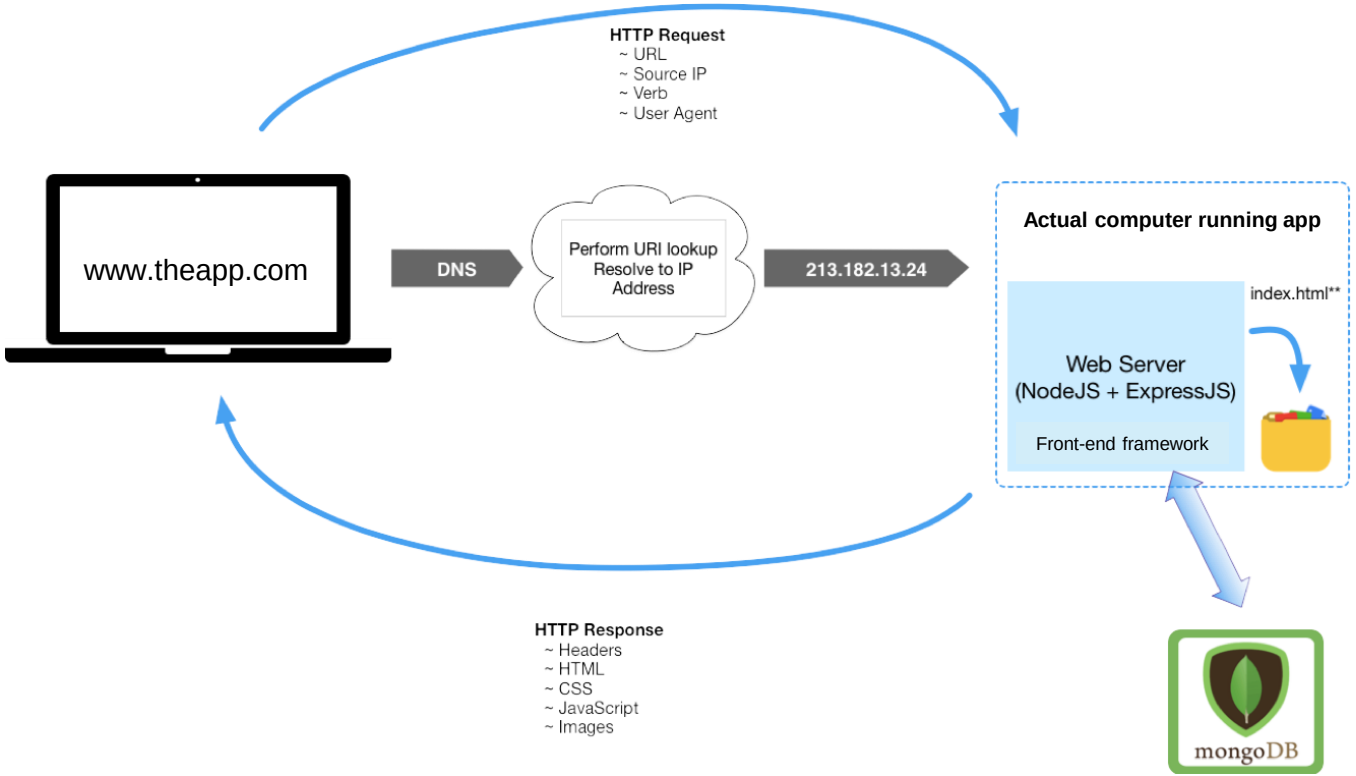
Pivoting mid-development

Principle: An agile, iterative process can allow a project to pivot in small or large degree, mid-process, without excessive cost of time or resources

Key aspects:

- Manage to meet business goals, due dates and budgets. Be willing to change requirements to fit these, not the other way around.
- Learn as you go; be adaptable to new or changing business needs that become clear only after development begins.
- Analyze existing implementations frequently to determine that they are meeting business goals.

Elements of a Full-Stack App and HTTP Request/Response Cycle



Group Discussion:

Looking at the “Reference Implementation” from an iterative approach

Group discussion considering the Reference Implementation from the following points of view:

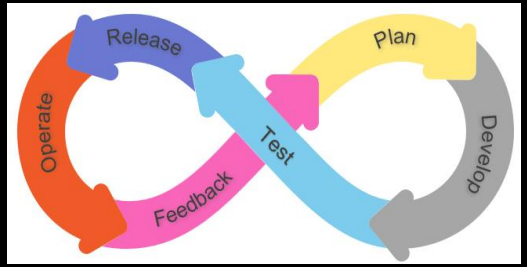
- Itemize the set of high-level features that will be needed to implement this application?
- Which features should be implemented first? Which features should ship first?
- How might your approach need to pivot, based on market or other factors?

Group Discussion:

Looking at the “Reference Implementation” from an iterative approach

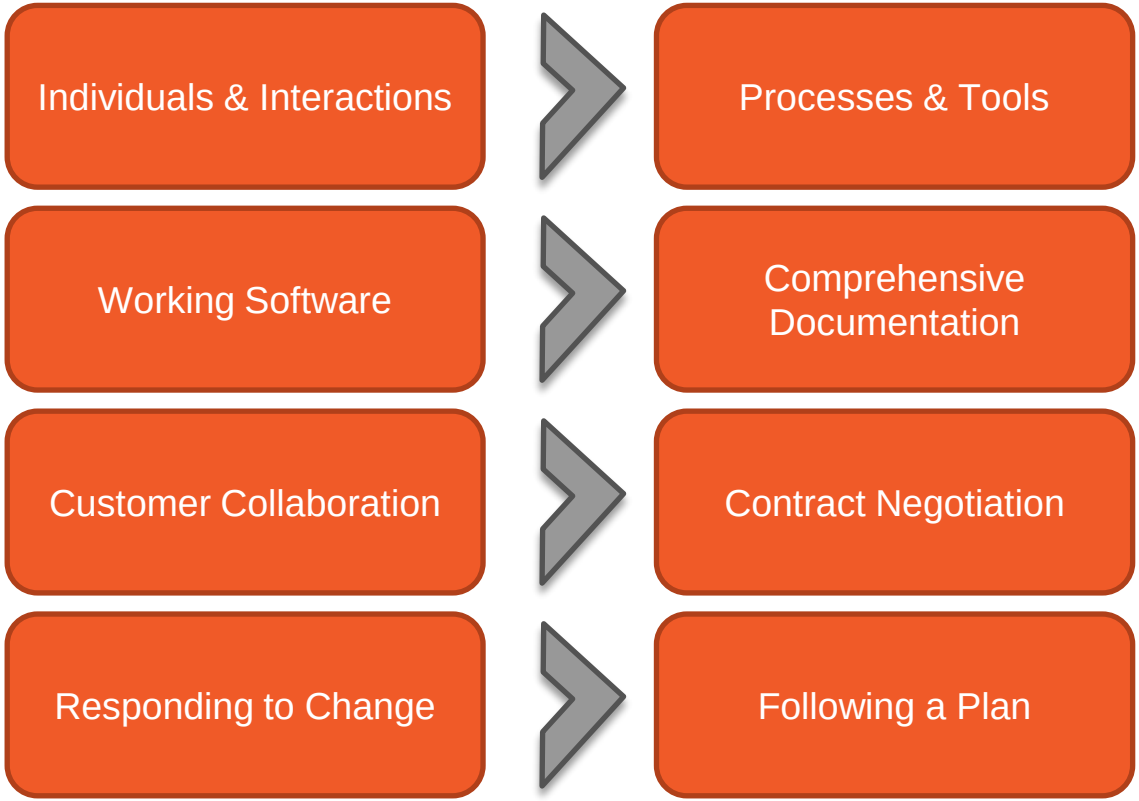
Group discussion considering the Reference Implementation from the following points of view:

- Based on your prioritization of the features that should be implemented/shipped first (from the previous discussion), what should your first two iterations look like?
- Include tasks for each phase (Plan, Develop, Test, Release, Operate)
- What are some practical ways to gather feedback on the features deployed as part of the first two iterations?



The Agile Manifesto and Best Practices

Agile Values



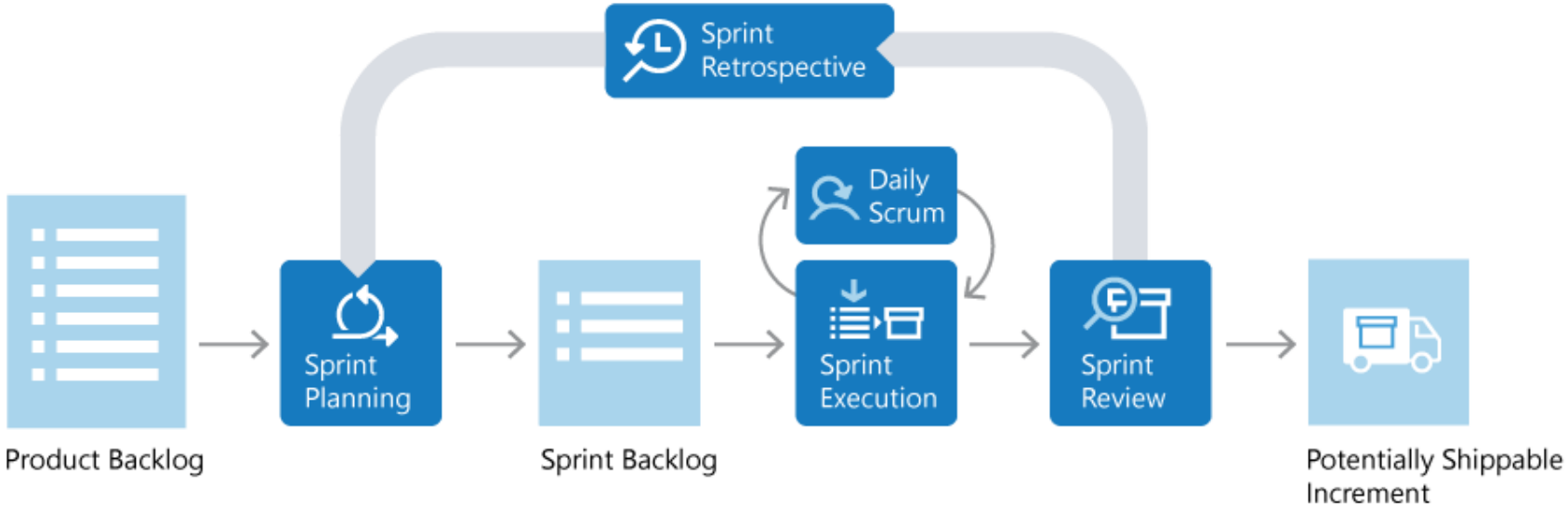
Modern Approaches to Agile Software Development

Several project management approaches implement Agile values in some way. Often, they bring in elements of other approaches.

Popular methodologies:

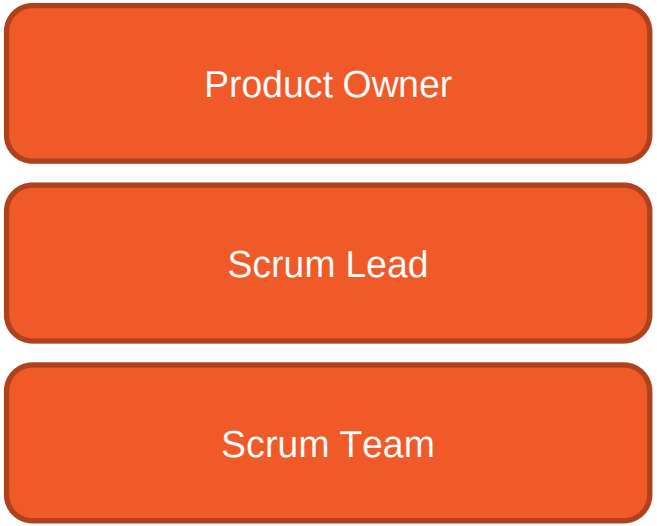
Kanban	Scrum	Scaled Agile Framework (SAFe)
• Uses visual boards to view & organize tasks • Uses elements of “just in time” lean manufacturing strategies • Emphasizes throughput	• Aligns closely with Agile values • Breaks down product development into iterative sprints • Makes use of exclusive roles (“Scrum Lead”, “Product Owner”) • Emphasizes constant communication	• Workflow and organizational patterns to help deploy Agile at scale • Can support large organizations

Diagram of Typical Scrum Sprint

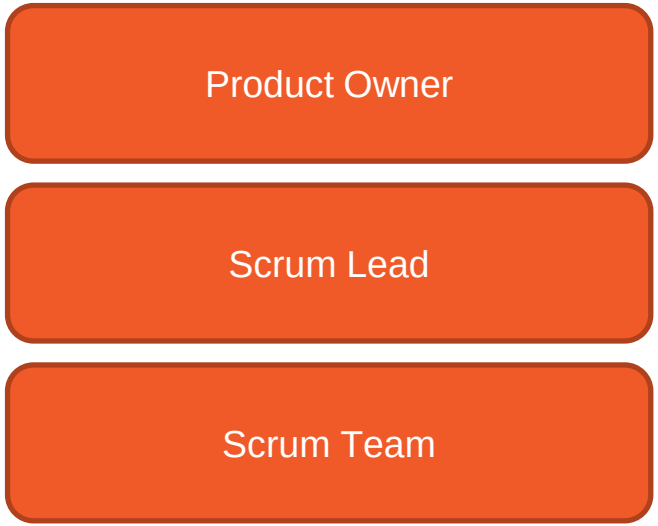


Source: https://docs.microsoft.com/en-us/devops/plan/what-is-scrum

Scrum Roles



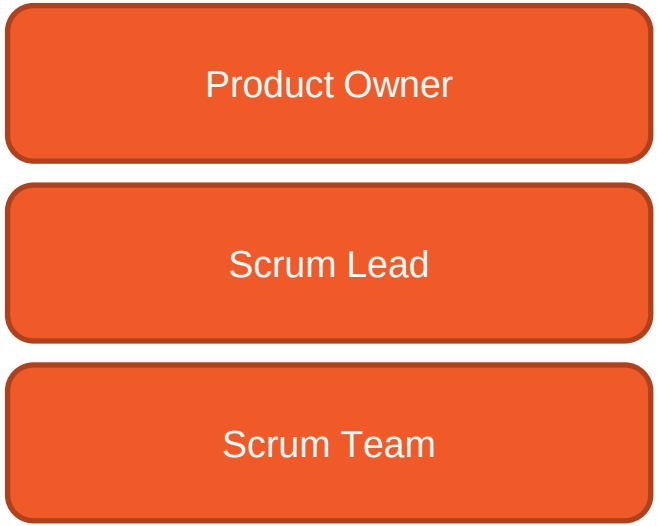
Scrum Roles



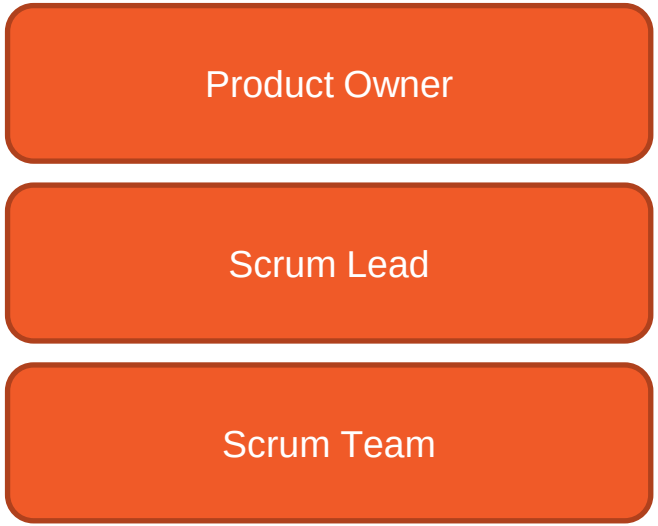
- Responsible for what team builds and why
- Keeps backlog up to date and in correct priority order

Scrum Roles

- Ensures that Scrum process is followed by team
- Responsible for the fidelity of the process
- Part coach, part team member, part cheerleader



Scrum Roles



- People building the product
- Responsible for product build (and associated quality)

Product Backlog

- Prioritized list of work the team will deliver
- Product owner manages – adds, changes, reprioritizes as needed
- Items at the “top of the list” get worked first

Backlog Grooming

- Regular team activity which allows group to verify that quality of defined sprint items remains high
- Team discusses each item and ensures that all required information to successfully complete is in place
- For example, has the item been given an adequate description and does the acceptance criteria provide team members with the information needed to assess “done”?
- Backlog item is assigned “points” representing relative level of effort to complete
- Points are not directly correlated to hours – instead, about amount of effort required for the task in comparison to previous tasks completed or other tasks planned
- There will likely be some number of points for items that is considered “too high” to complete as a single item
- Provides threshold for determining when to divide a large task into smaller parts

Sprint Planning

- Team chooses items from the backlog to target for upcoming sprint (based on prioritization by Product Owner)
- Intended to represent amount of work team thinks they can complete within the sprint
- Items in the sprint may be broken down into a set of more granular tasks
- Historical data from previous sprints can help a team determine its velocity (i.e., how many points the team can complete in an iteration)
- Results in a “line” for the sprint within the backlog – items above the line are “in”, everything else is pushed to future iteration
- If something below the line needs to move above, it will need to be swapped with an existing task to ensure capacity to deliver

Sprint Execution

- Team executes on the items identified for inclusion in the iteration
- Short daily meetings are held among team members (called daily Scrums) – usually no more than 15 minutes
- Each member briefly describes what they worked on the previous day, what they're planning to work on today, and any "blockers" (issues preventing progress)
- As part of daily review, a sprint burndown chart can be used to assess whether or not the team is on track to complete what was committed to

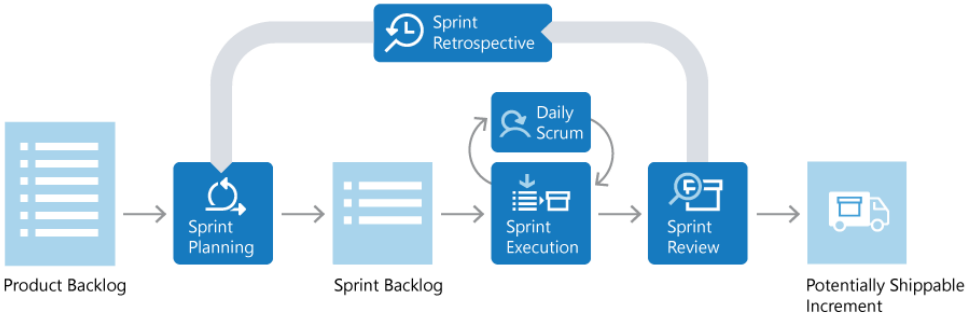
Sprint Review

- At the end of the iteration, team demonstrates for stakeholders what’s been accomplished during the sprint
- Goal is to show incremental value

Sprint Retrospective

- Team reflects on what went well during the last sprint
- Just as important (if not more so) is identifying specific areas for improvement to take into future sprints
- Outcome should include a set of action items to help steward improvement

Group Reflection



- Where does architecture fit within each “ceremony”?
- How do we ensure architecture is not an afterthought but an active consideration during sprint planning & execution?

Architecting for the Future

Architecting for the Future

- When we architect and build an application at a “point in time”, we hope that the application will continue to be utilized to provide the value for which it was originally built
- Since the “only constant is change” (a quote attributed to Heraclitus of Ephesus), we have to expect that the environment in which our application “lives and works” will be dynamic
- Change can come in the form of business change (change to business process), the need to accommodate innovation and ongoing advancement in technology
- Often, the speed at which we can respond to these changes is the difference between success and failure

Architecting for the Future

In order to ensure that we can respond to change “at the speed of business”, we need to build our systems according to best practices and good design principles:

- Business-aligned design
- Separation of concerns
- Loose coupling
- Designing for testability

Business-Aligned Design

- Build systems that use models and constructs that mirror the business entities and processes that the system is intended to serve
- Drive the design of the system and the language used to describe the system based around the business process not the technology
- AKA Domain Driven Design
- Results in a system built out of the coordination and interaction of key elements of the business process – helps to ensure that the system correlates to business value
- Also helps business and technology stakeholders keep the business problem at the forefront

Separation of Concerns

- Break a large, complex problem up into smaller pieces
- Drive out overlap between those pieces (modules) to keep them focused on a specific part of the business problem and minimize the repeat of logic
- Logic that is repeated, and that might change, will have to be changed in multiple places (error prone)
- Promotes high cohesion and low coupling (which we will talk about in a minute)
- Solving the problem becomes an exercise in “wiring up” the modules for end-to-end functionality and leaves you with a set of potentially reusable libraries

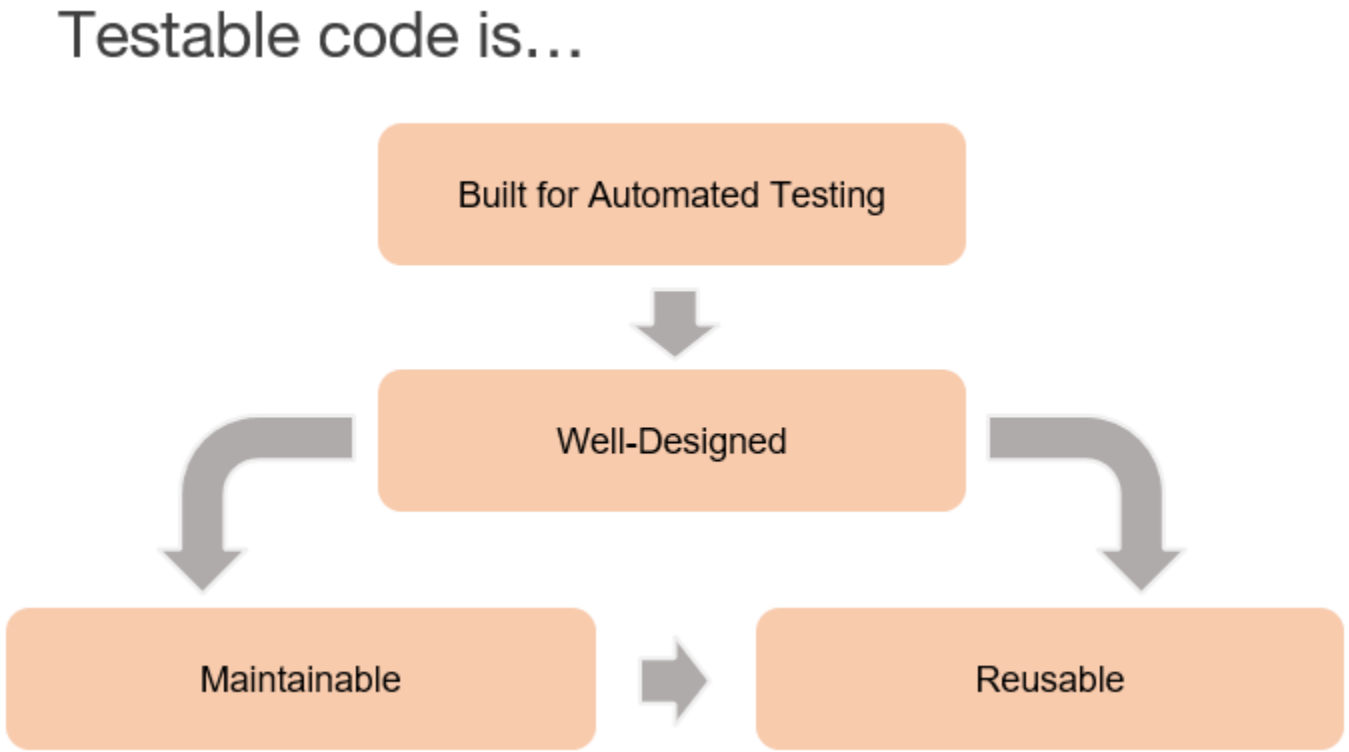
Loose Coupling

- Coupling between components or modules in a system causes problems, especially for maintaining the system over time
- System components that are tightly coupled, are more difficult to change (or enhance) – changes to one part of the system may break one or more other areas
- With coupling, you now must manage the connected components as a unit instead of having the option to manage the components in different ways (e.g. production scalability)
- Makes the job of unit testing the components more difficult because tests must now account for a broader set of logic and dependencies (e.g. tight coupling to a database makes it difficult to mock)

Designing for Testability

- Practicing the previous principles helps lead to a system that is testable
- Testability is important because it is a key enabler for verifying the quality of the system – at multiple stages along the Software Development Lifecycle (SDLC)
- When building a system, quality issues become more expensive to correct the later they are discovered in the development lifecycle – good architecture practices help you test early and often
- Ideally, testing at each stage will be automated as much as possible in support of quickly running the tests as and when needed

Designing for Testability



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

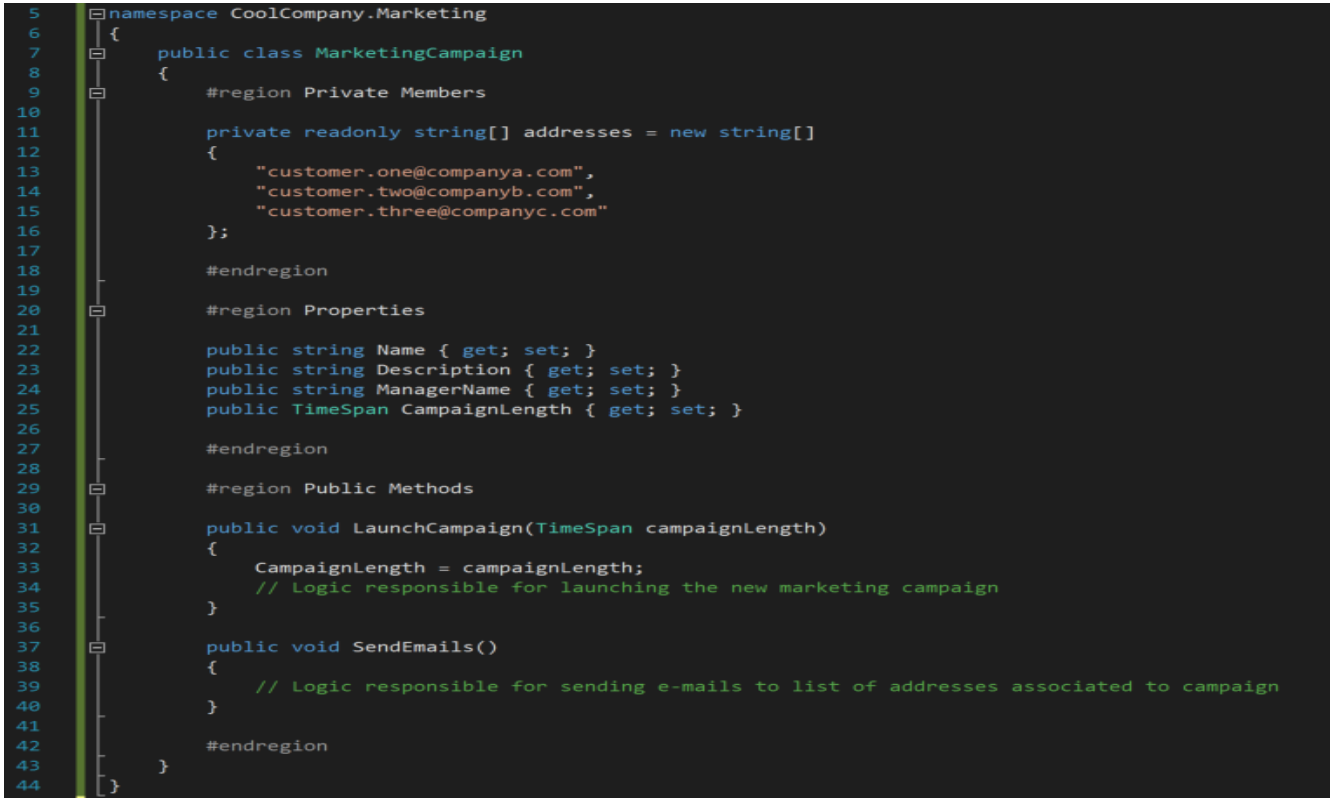
Single Responsibility Principle (SRP)

Single Responsibility Principle (SRP)

Why important?

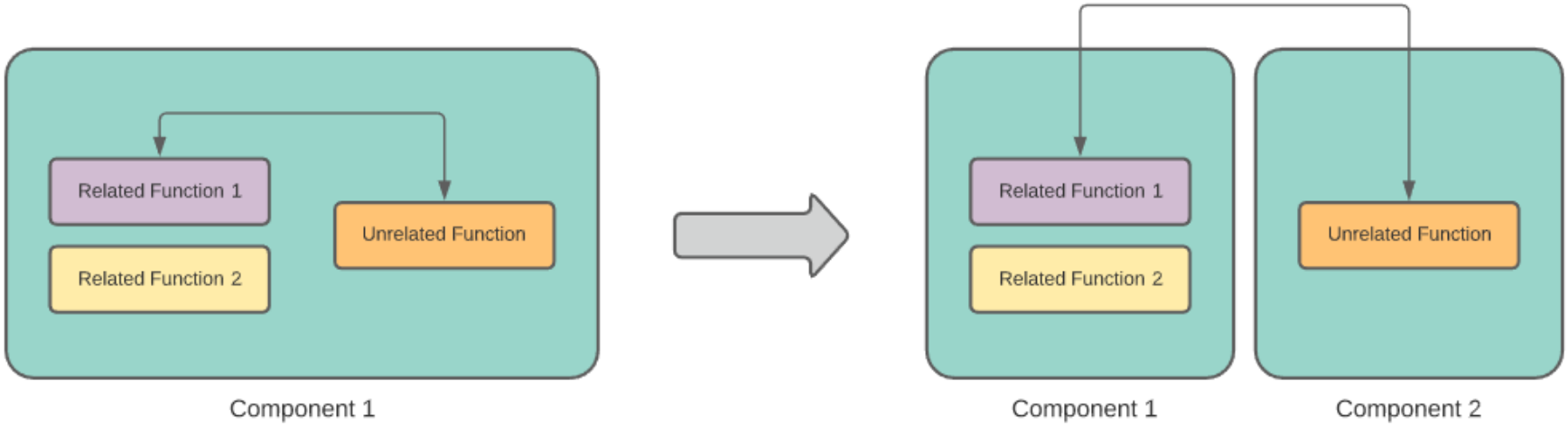
- Related functions organized together breed understanding (logical groupings) – think *cohesion*
- Multiple, unrelated functionalities slammed together breed coupling
- Reduced complexity (cleaner, more organized code)
- Promotes smaller code modules
- Reduced regression – tension of change

Single Responsibility Principle (SRP)



```
5   namespace CoolCompany.Marketing
6   {
7       public class MarketingCampaign
8       {
9           #region Private Members
10
11          private readonly string[] addresses = new string[]
12          {
13              "customer.one@companya.com",
14              "customer.two@companyb.com",
15              "customer.three@companyc.com"
16          };
17
18          #endregion
19
20          #region Properties
21
22          public string Name { get; set; }
23          public string Description { get; set; }
24          public string ManagerName { get; set; }
25          public TimeSpan CampaignLength { get; set; }
26
27          #endregion
28
29          #region Public Methods
30
31          public void LaunchCampaign(TimeSpan campaignLength)
32          {
33              CampaignLength = campaignLength;
34              // Logic responsible for launching the new marketing campaign
35          }
36
37          public void SendEmails()
38          {
39              // Logic responsible for sending e-mails to list of addresses associated to campaign
40          }
41
42          #endregion
43      }
44  }
```

Single Responsibility Principle (SRP)



Single Responsibility Principle (SRP)

How to practice?

- When building new modules (or refactoring existing), think in terms of logical groupings
- Look for “axes of change” as points of separation
- Build new modules to take on new entity or service definitions – provides abstraction
- Structure clean integrations between separated modules
- Use existing tests (or build new ones) to verify a successful separation

EXAMPLE:

Single Responsibility Principle (SRP)

Review the example available at https://www.geeksforgeeks.org/single-responsibility-principle-in-java-with-examples/

LAB 01:

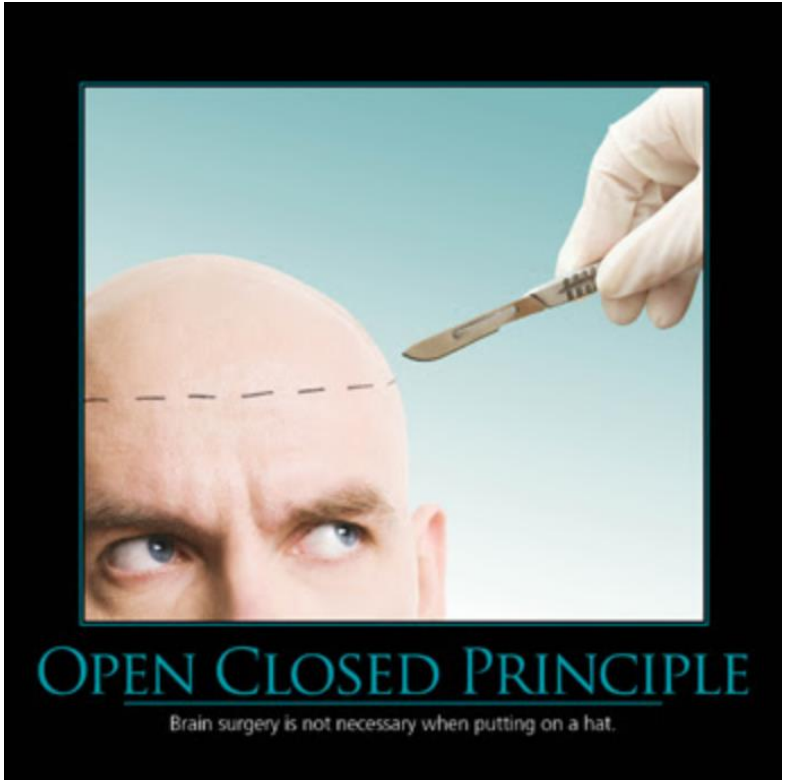
Single Responsibility Principle (SRP)

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab01

Open-Closed Principle (OCP)

Software entities should be open for extension but closed for modification

Open-Closed Principle (OCP)



Open-Closed Principle (OCP)

Why important?

- Our systems need to be able to evolve
- We need to be able to minimize the impact of that evolution

Open-Closed Principle (OCP)

How to practice?

- When building new modules (or refactoring existing), leverage abstractions
- Use the abstractions as levers of extension
- Use existing tests (or build new ones) to verify the abstractions

EXAMPLE:

Open Closed Principle (OCP)

Review the example available at https://www.geeksforgeeks.org/open-closed-principle-in-java-with-examples/

LAB 02:

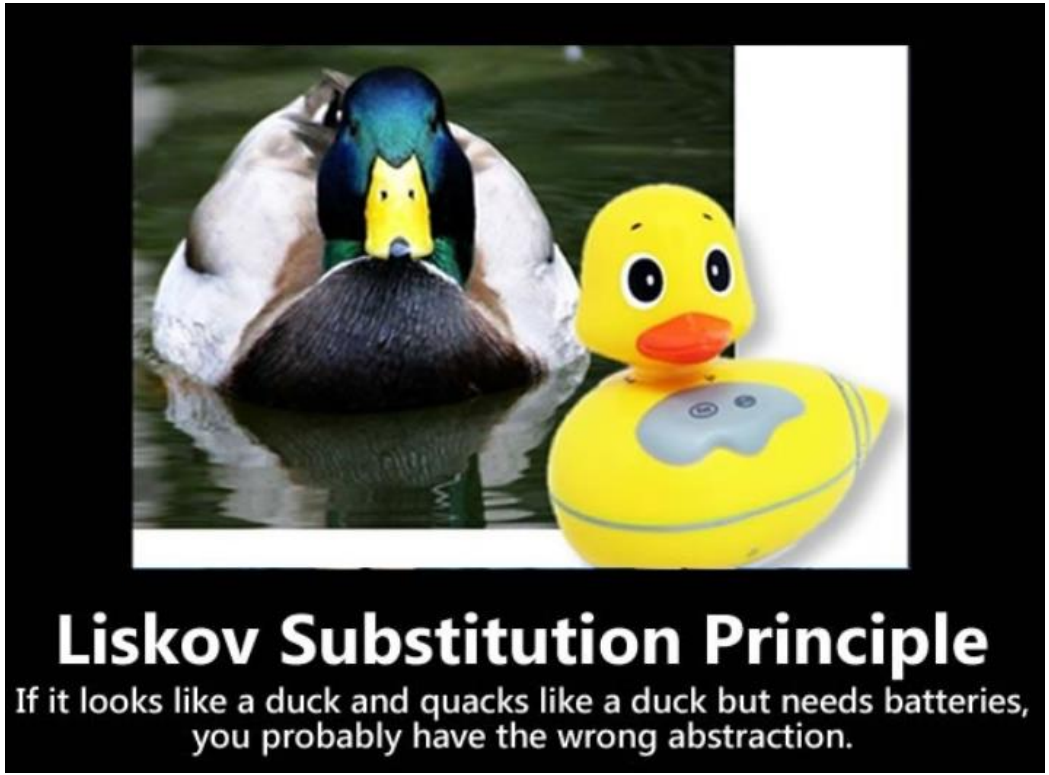
Open Closed Principle (OCP)

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab02

Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types

Liskov Substitution Principle (LSP)



Liskov Substitution Principle (LSP)

Why important?

- Want to be able to use the abstractions created by OCP to extend existing functionality (vs. modify it)
- Especially useful when multiple variants of a type need to be processed as a single group
- Promotes looser coupling between our modules
- Without it, we may have to include if/else or switch blocks to route our logic
- Or keep adding parameters/properties for new but related types

Liskov Substitution Principle (LSP)

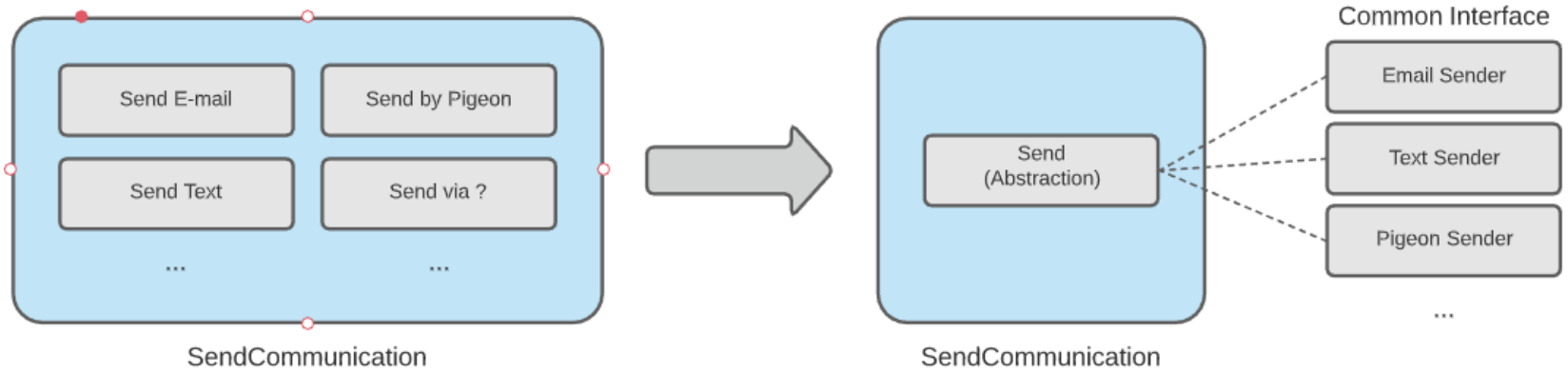
How to practice?

- Module members should reference the abstractions in consuming code
- Look for ways to encapsulate type-specific (or type-aware) logic in the type instead of in the code using the type
- Use existing tests (or build new ones) to verify our ability to effectively substitute

OCP & LSP

```
14
                0 references
15              public class CommunicationAgent
16              {
                    0 references
17                  public void SendCommunication(CommunicationType communicationType)
18                  {
19                      if (communicationType == CommunicationType.Email)
20                      {
21                          // Call to another component that knows about sending e-mail
22                      }
23                      else if (communicationType == CommunicationType.Text)
24                      {
25                          // Call to another component that knows about sending texts
26                      }
27                      else if (communicationType == CommunicationType.Pigeon)
28                      {
29                          // Call to another component that knows about sending messages by pigeon
30                      }
31                      else
32                      {
33                          // Error or default behavior - unrecognized communication type
34                      }
35                  }
36              }
37          }
38
```

OCP & LSP


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

EXAMPLE:

Liskov Substitution Principle (LSP)

Review the example available at https://dzone.com/articles/the-liskov-substitution-principle-with-examples

LAB 03:

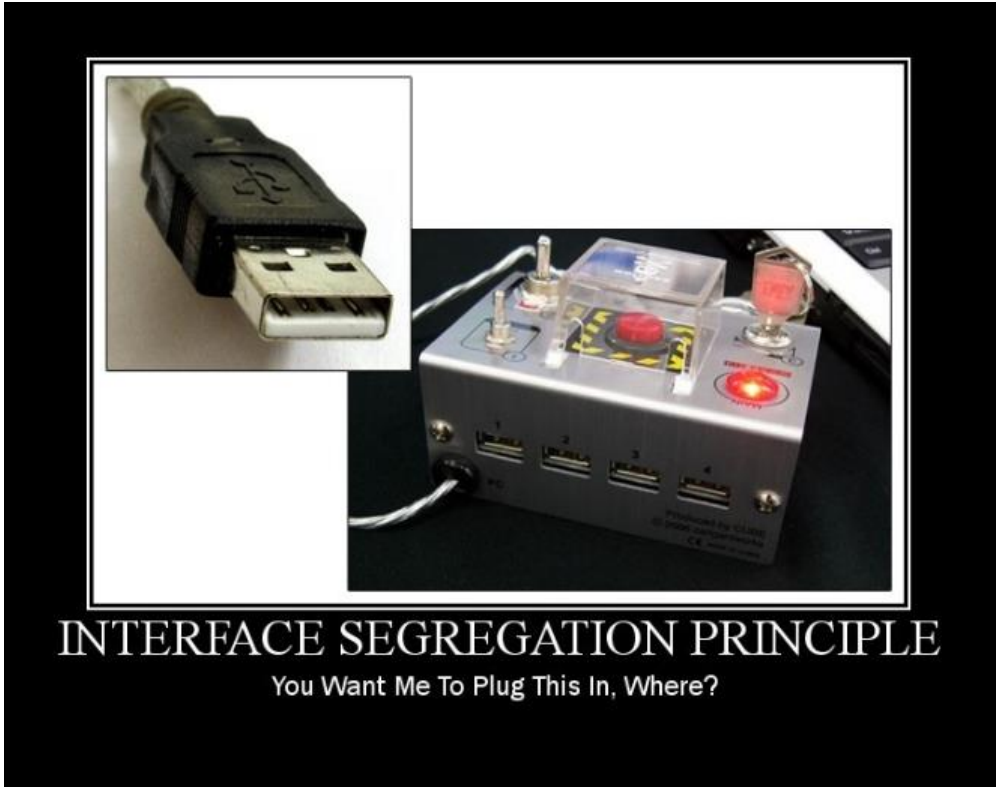
Liskov Substitution Principle (LSP)

https://github.com/KernelGamut32/advanced-software-architecture-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)



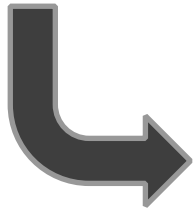
Interface Segregation Principle (ISP)

Why important?

- By following SRP, OCP and LSP, we can build a set of cohesive abstractions enabling reuse in multiple clients
- However, sometimes the abstractions we need are not cohesive (even though they may seem like it at first)
- We need a mechanism for logical separation that still supports combining functions together in a loosely-coupled way
- Otherwise, we'll see "bloating" in our abstractions that can cause unintended/unrelated impact during normal change

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 }
```

Interface Segregation Principle (ISP)

How to practice?

- In your abstractions, don’t force functions together that don’t belong together (or that you might want to use separately)
- Leverage delegation in the implementation of those abstractions to support variance
- Use OCP to bring together additional sets of features in a cohesive way (that still adheres to SOLID)
- Use existing tests (or build new ones) to verify aggregate features

EXAMPLE:

Interface Segregation Principle (ISP)

Review the example available at https://www.javabrahman.com/programming-principles/interface-segregation-principle-explained-examples-java/

LAB 04:

Interface Segregation Principle (ISP)

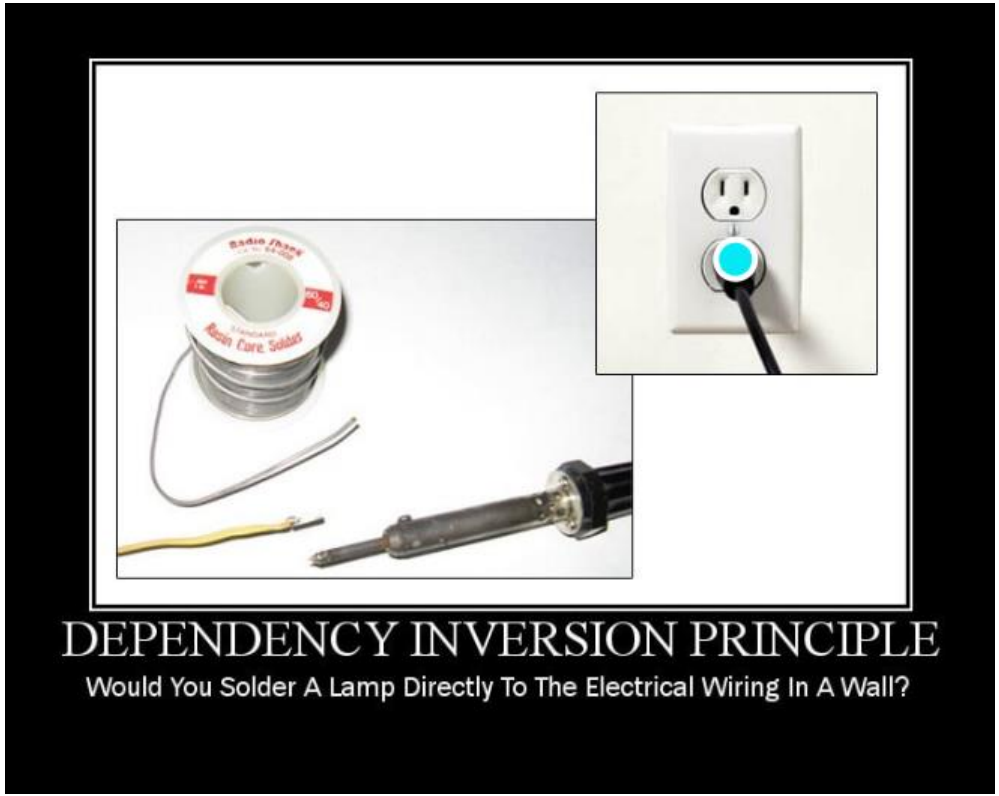
https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab04

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)



Dependency Inversion Principle (DIP)

Why important?

- As with all the SOLID principles, we want to build reusable but loosely-coupled code modules
- We don’t want to limit reuse to lower-level utility classes only
- If higher-level modules are tightly-coupled to and dependent on low-level modules, change impact can cascade up
- The change impact can be transitive (flowing through multiple layers in-between)
- There are patterns and utility libraries built to enable

Dependency Inversion Principle (DIP)

How to practice?

- As with the other principles, use (and depend on) abstractions
- Use mechanisms like dependency injection and IoC (Inversion of Control) to build looser coupling between logic providers and consumers
- Build layering into your architectures and limit references to the same or immediately adjacent layer only
- Keep ownership of abstractions with the clients that use them or, even better, in a separate namespace/library
- Use existing tests (or build new ones) to verify functionality in each layer and use mocking techniques to isolate testing

EXAMPLE:

Dependency Inversion Principle (DIP)

Review the example available at https://www.geeksforgeeks.org/dependecy-inversion-principle-solid/

LAB 05:

Dependency Inversion Principle (DIP)

https://github.com/KernelGamut32/advanced-software-architecture-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

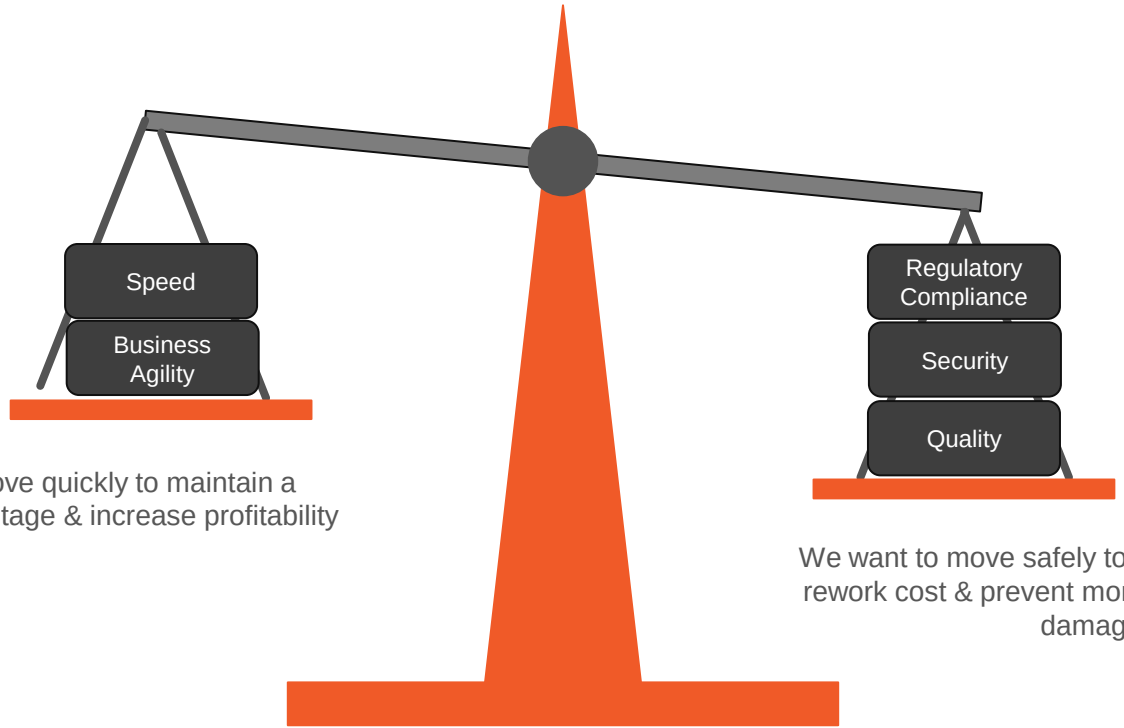
Reduced fear

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

What if we wrote our tests *first?*

Test Driven Development reverses the usual

Devs must devise a test that will verify correct operation of a function *before they even write the function*

This forces you to write functions that “do one thing, and do it well”

Test Driven Development – basic sequence

Write a unit test for the function you're about to create

Run the test – it should fail

Write the function

Test it

If it fails, re-examine your test and change it if your logic was wrong

Otherwise, refactor your function until it passes

Write another test - repeat

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

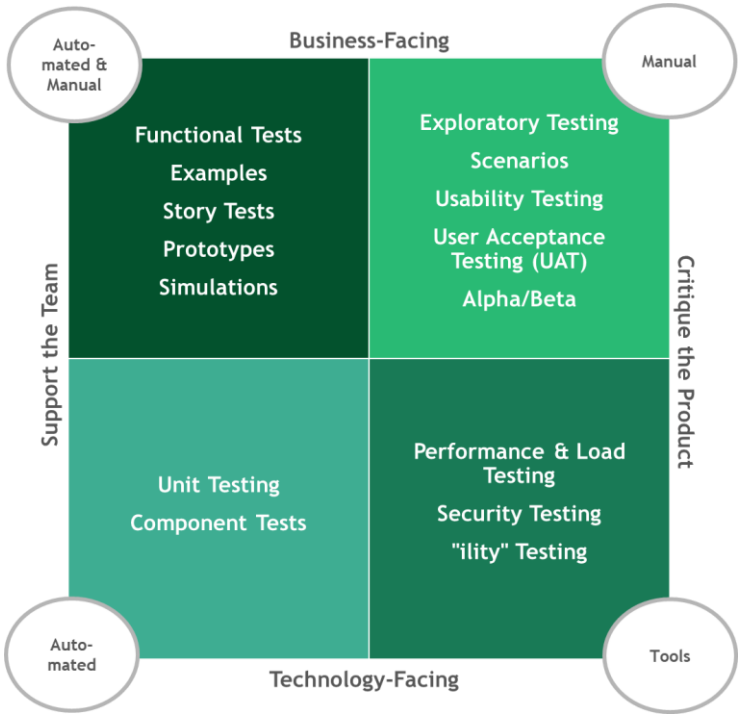
What are some of the challenges you face with software testing in your jobs (i.e., why don’t we test as much as we want to or should)?

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?”

Testing Strategies

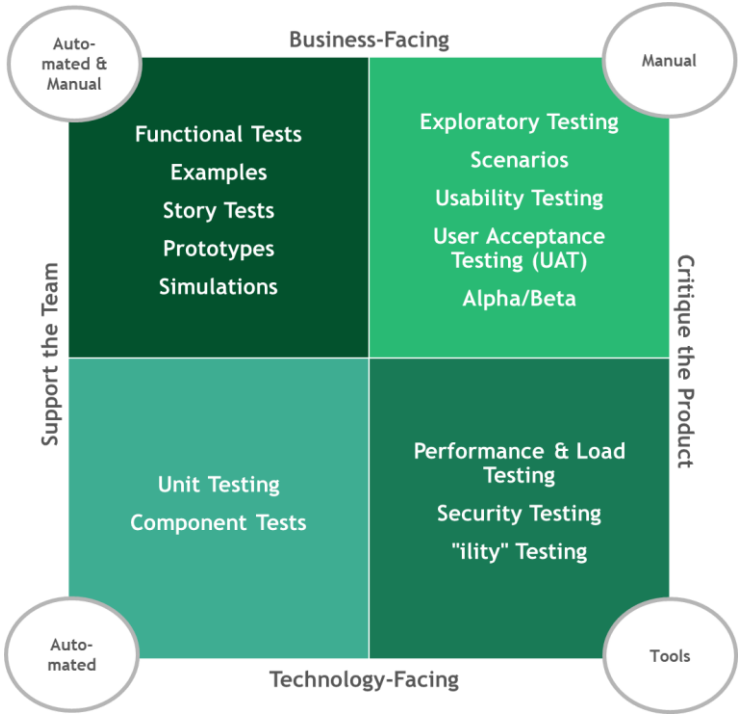
Using the Agile Testing Quadrant



Testing Quadrant

Business Facing

- Confirm the external quality of the product



Support the Team

- Focused on supporting the teams that underpin the product & its delivery
- Goal is to help guide the quality of the product & prevent functional defects from making their way into production

Critique the Product

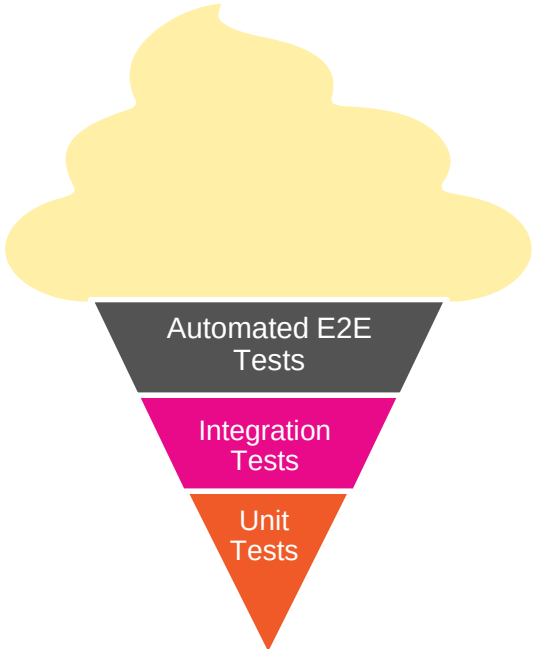
- Focused on evaluating overall product quality
- How will software that meets the functional requirements perform (on multiple levels) when deployed to production?
- Ensuring that Quality of Service (QoS) are met

Technology Facing

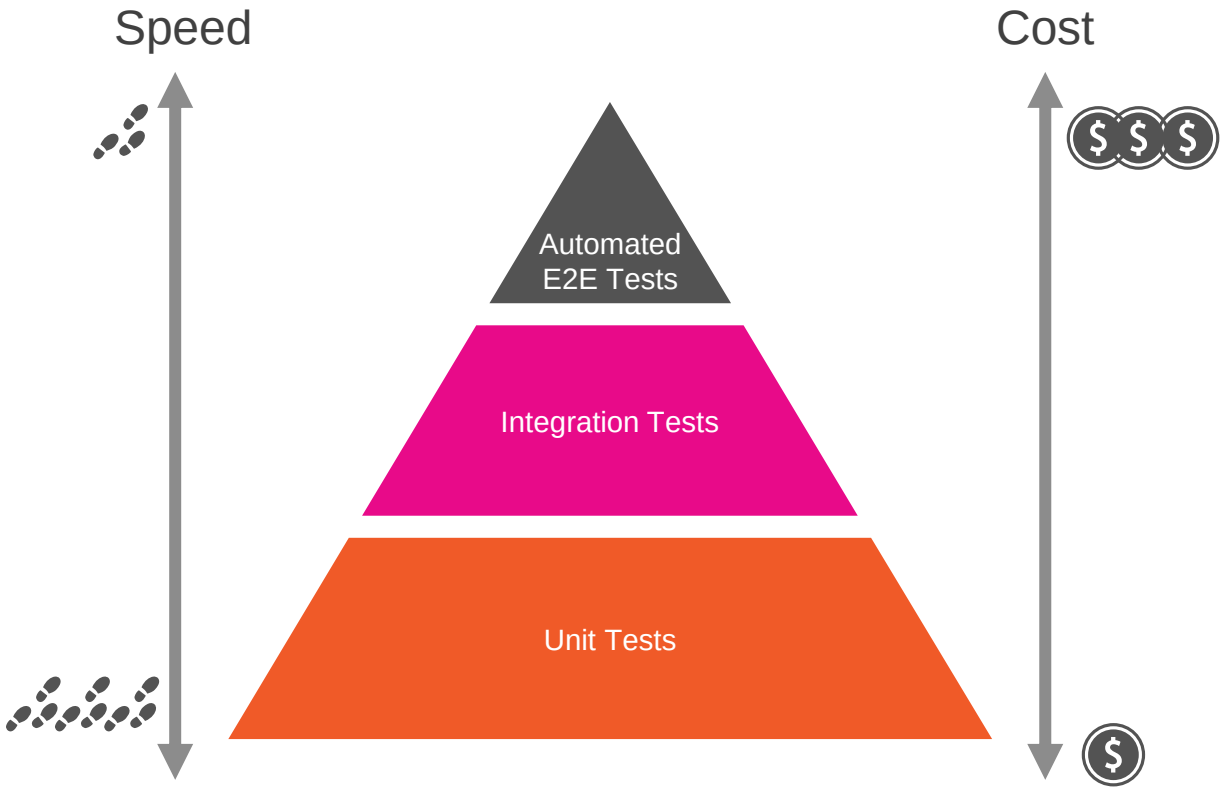
- Maintain the internal quality of the application

Source - https://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/

At times, without vigilance, we may find that our software testing resembles an ice cream cone rather than a pyramid

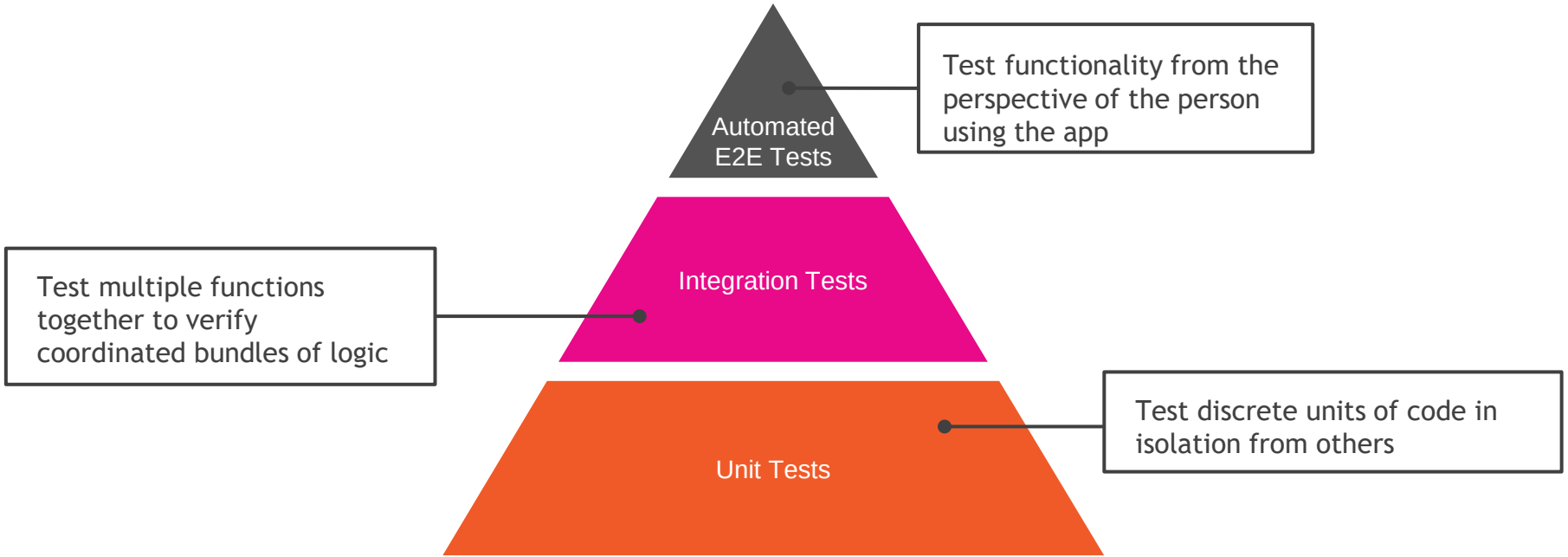


Test Pyramid



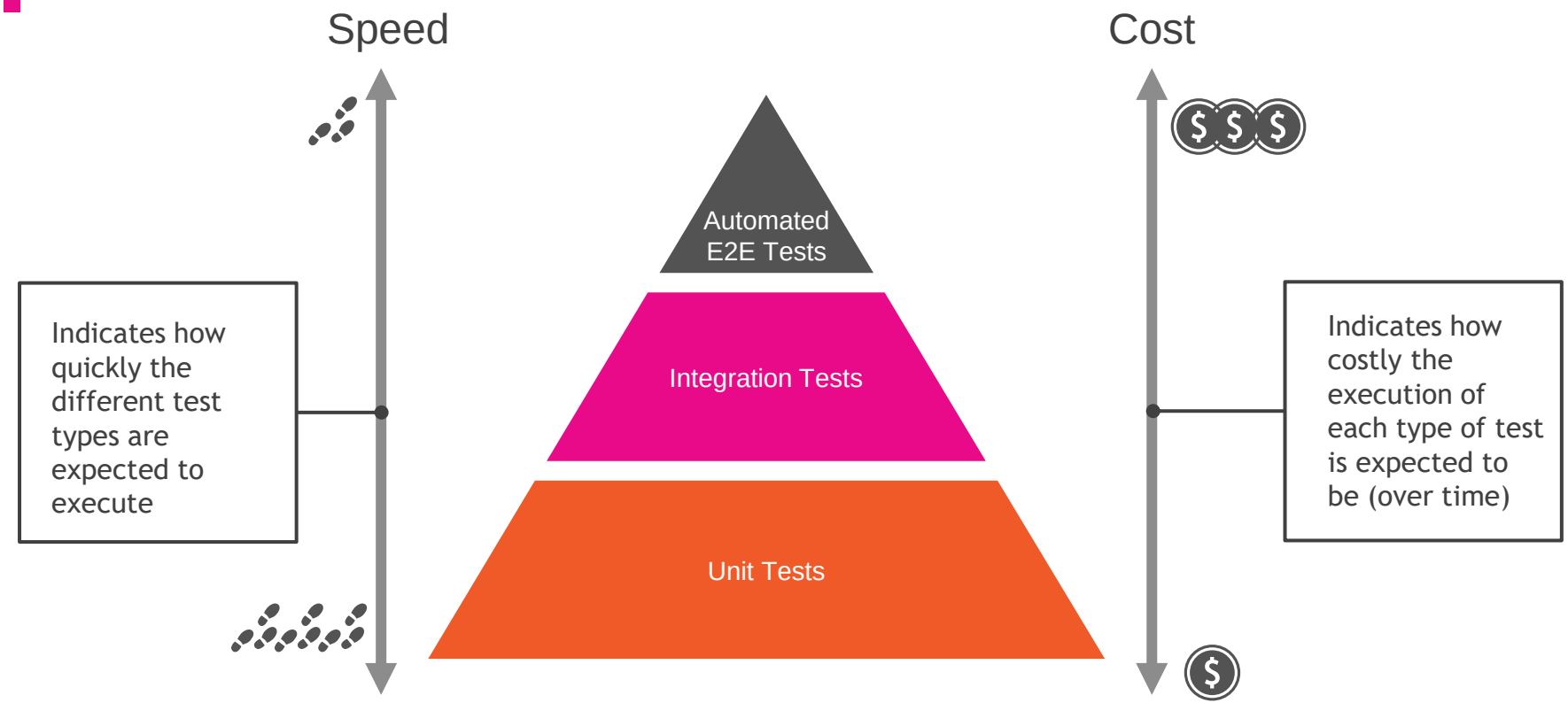
Source - https://martinfowler.com/bliki/TestPyramid.html

Test Pyramid



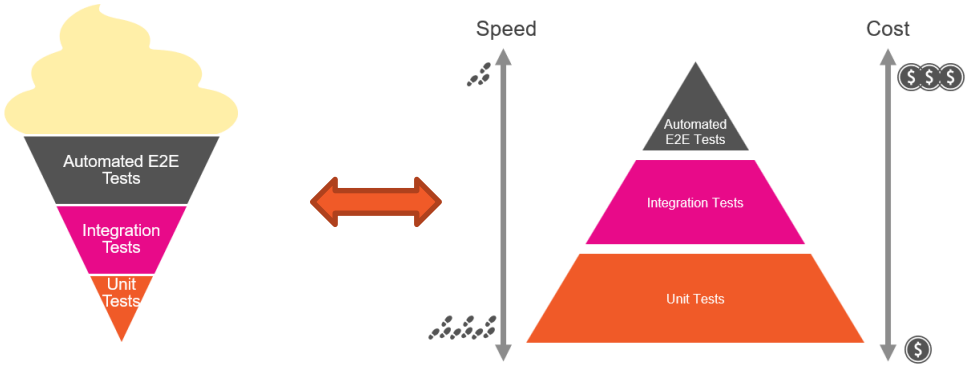
Source - https://martinfowler.com/bliki/TestPyramid.html

Test Pyramid



Source - https://martinfowler.com/bliki/TestPyramid.html

Reflection



1. What are some of the root causes that you think could cause a shift from the pyramid to the ice cream cone?

1. During project execution, what are some specific warning signs we can look for that will alert us to a trend towards that shift?

1. What are some practical steps we can take to move from the ice cream cone to a proper testing pyramid?

Managing Software Quality

Using Test Coverage to evaluate the completeness & effectiveness of our tests

The What

- Measurement mechanisms are built into many languages & development frameworks
- These tools analyze source code under test against automated test runs, identifying source code that is covered (exercised by one or more tests) vs. not covered (excluded from any test)

The How

- Ranges from 0 – 100%, and the higher the number, the better the coverage
- A low test coverage percentage indicates a deficiency in automated testing
- Test coverage measures are often integrated into CI/CD pipelines, failing the build if a minimum level of coverage has not been achieved

Discussion | What is the “correct” target percentage for unit test coverage?

Wrangling “Technical Debt”

- Involves borrowing from the future to address the immediate
- Manifests as choosing to go with a sub-optimal solution to move a project forward, instead of spending the time to build out the optimal approach
- Results in higher compounding costs in the long-run (e.g., future development cost, long-term maintainability, speed, quality, etc.)

20-30%

of sprint capacity, ideally, should be allocated for bug fixes, automation improvement, POCs, etc.

Tactics to mitigate technical debt:

1. Find opportunities to pay down debt as part of a future enhancement

1. Remove debt when modernizing a legacy system – don’t simply do a “balance transfer” when building the replacement

1. Embrace pay down of technical debt as a regularly-planned activity within your sprints (if possible)

LAB 06:

Testing Classes

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab06

LAB 07:

Testing Classes

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab07

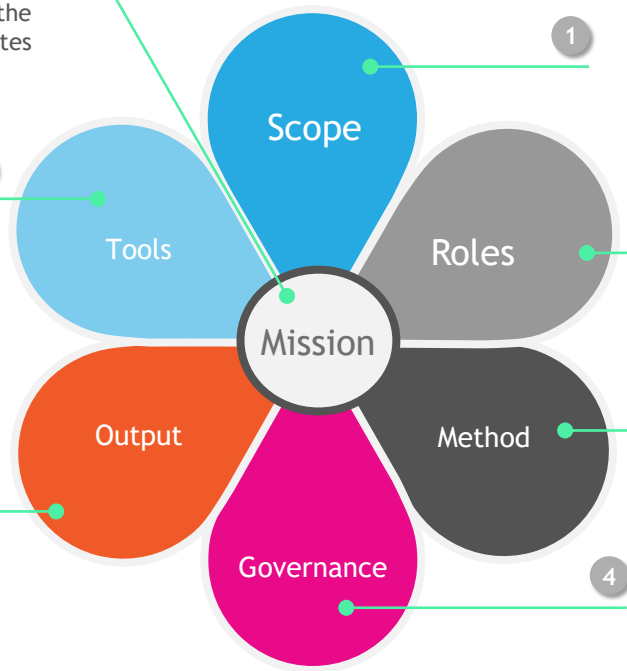
LAB 08:

TDD

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab08

Solution & Evolutionary Architecture

Understanding 7 dimensions of architecture



0 **Mission: purpose, goals, objectives and values of Architecture**

Describe why Architecture is important and the value it generates

1 **Definitions of Architecture**

Describe the differences between Enterprise, Solution and Domain architecture

2 **Architecture Roles**

Describe the different architecture roles and associated responsibilities

3 **Ways of working**

How architects relate and cooperate among them, and with other key roles

4 **Definition and explanation of Architecture Governance**

Describe how architecturally significant decisions are made

5 **Architectural deliverables & artifacts**

Architecture outputs, like guidelines, designs and architectural recommendations

6 **Software solutions enabling the architecture practice**

Clarify and encourage usage of tools in executing architecture management

The Mission

Architecture seeks to build out capabilities that enable the business vision of the organization through technology

Architecture seeks to avoid different (sometimes substandard) approaches to solving the same business or technical problem because that has a high cost & high risk

Architecture seeks to help an organization minimize “accidental complexity”

Architecture seeks to provide guidance to the larger organization on standards & best practices for optimal project execution

Architecture, in concert with other teams, seeks to provide oversight of an organization’s technology landscape to ensure that it improves rather than degrades over time

Solution Architecture vs. Application Architecture

TOGAF defines the terms “Solution Architecture”, and “Application Architecture” as follows:

3.69 Solution Architecture

A description of a discrete and focused business operation or activity and how IS/IT supports that operation.

Note:
A Solution Architecture typically applies to a single project or project release, assisting in the translation of requirements into a solution vision, high-level business and/or IT system specifications, and a portfolio of implementation tasks.

3.3 Application Architecture

A description of the structure and interaction of the applications as groups of capabilities that provide key business functions and manage the data assets.

Source: https://pubs.opengroup.org/architecture/togaf9-doc/arch/chap03.html

Solution Architecture vs. Application Architecture

- Often described as a function of scope
- Am I focused on one or more specific solutions or services? (Solution Architecture)
- Or, am I focused on the discipline of architecting applications to enable business & technical capabilities? (Application Architecture)

In alignment with the other types of architectural roles, the goal is consistency and coordination

- ➢ Can we use a specific project to move the organization forward architecturally?
- ➢ Does the architecture for this solution (or this capability) align with our enterprise standards & best practices?

Architecture Design in Practice

Methodology

Methodology

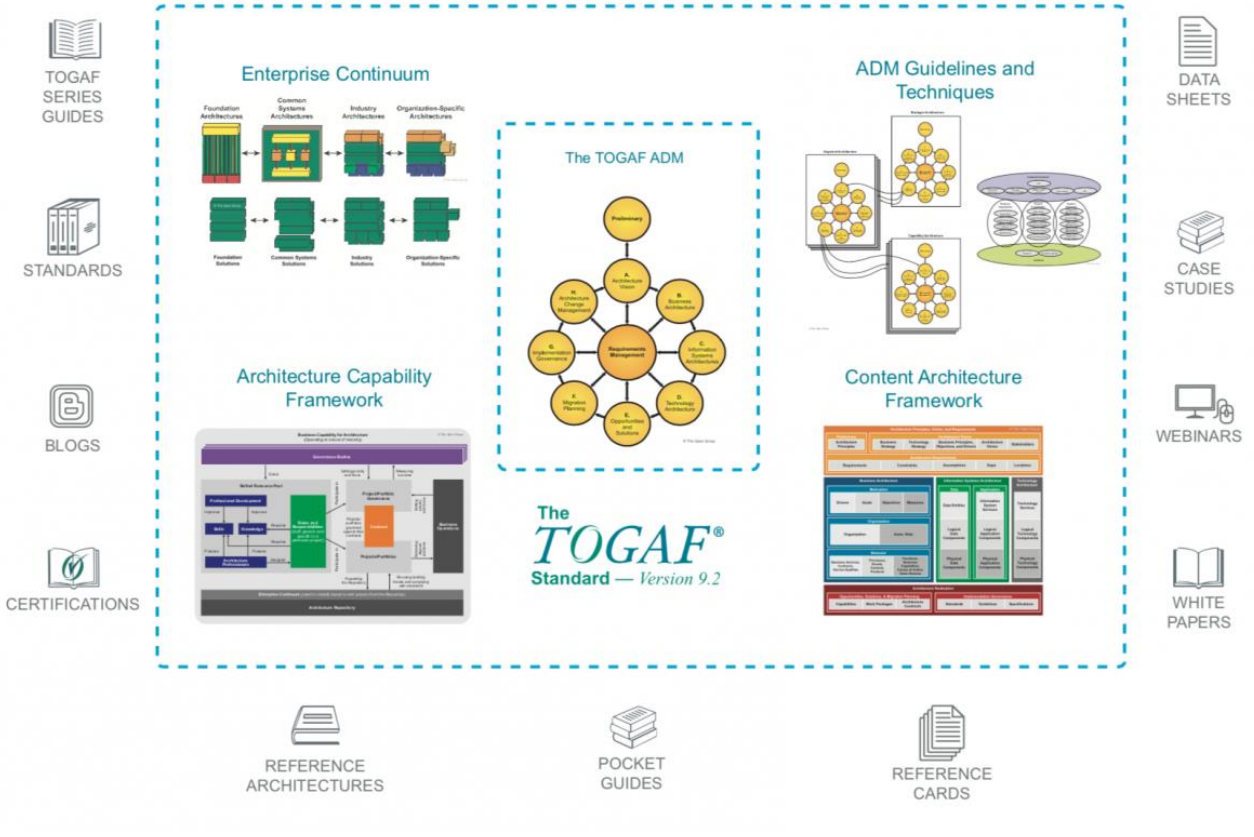
What is the “list” of concerns we should be focused on to ensure organizational success?

How do we effectively coordinate those multiple architectural concerns across the enterprise?

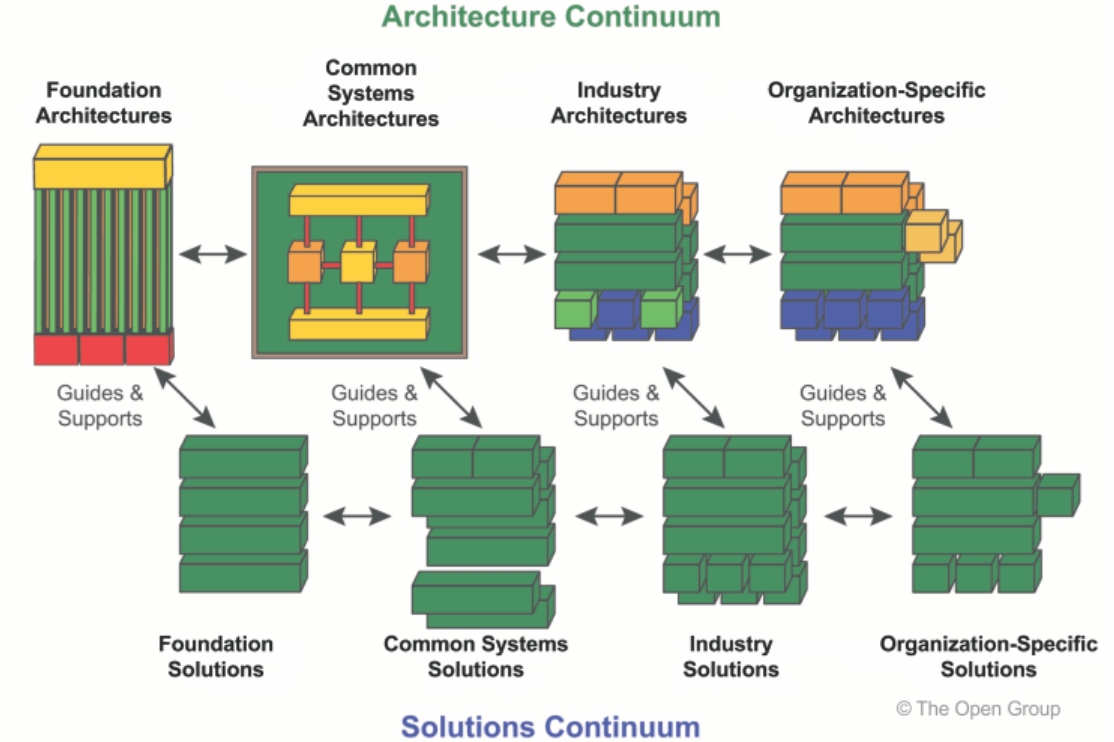
Is there a proven & repeatable process that we can follow to ensure we are documenting & managing to the right areas of concern?

Does this process support development of the architectural capability over time as our organization matures?

Enter TOGAF


TOGAF SERIES GUIDES
STANDARDS
BLOGS
CERTIFICATIONS
Enterprise Continuum
The TOGAF ADM
ADM Guidelines and Techniques
Architecture Capability Framework
Content Architecture Framework
The TOGAF® Standard — Version 9.2
DATA SHEETS
CASE STUDIES
WEBINARS
WHITE PAPERS
REFERENCE ARCHITECTURES
POCKET GUIDES
REFERENCE CARDS

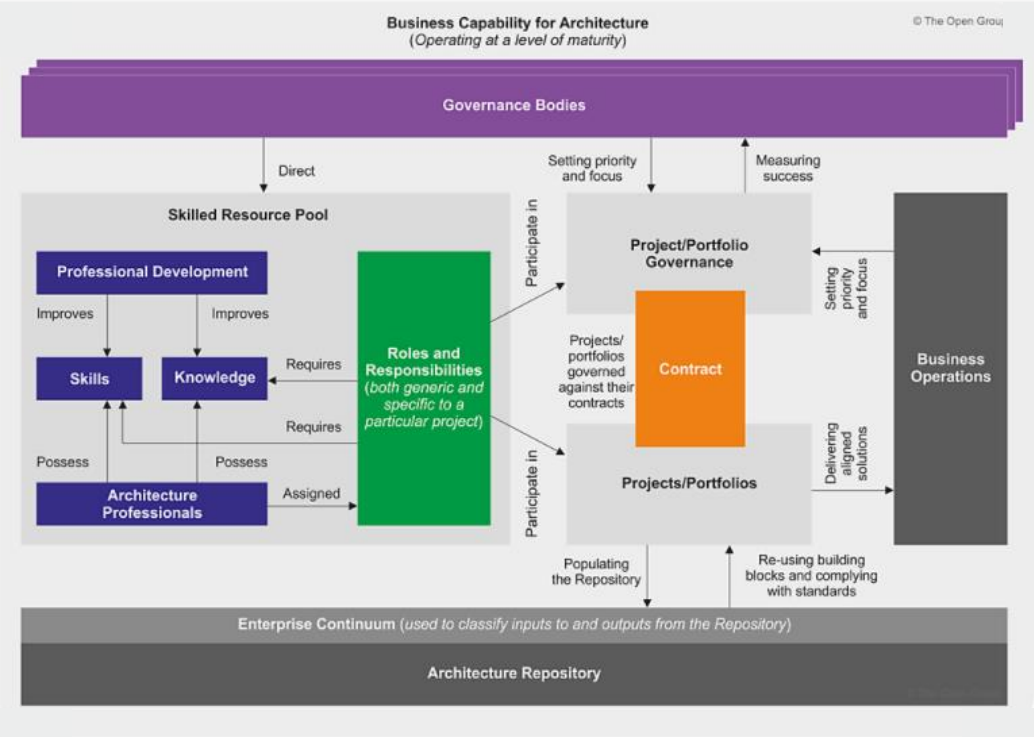
Enterprise Continuum



- Enterprise Continuum is made up of Architecture and Solutions Continuums
- Provides context for the full set of architecture & solution assets that apply across the enterprise
- Architecture Continuum classifies architecturally significant assets (from generic to specific)
- Solutions Continuum classifies specific solutions that implement and achieve architectural intent

Source: https://pubs.opengroup.org/architecture/togaf91-doc/arch/chap39.html

Architecture Capability Framework

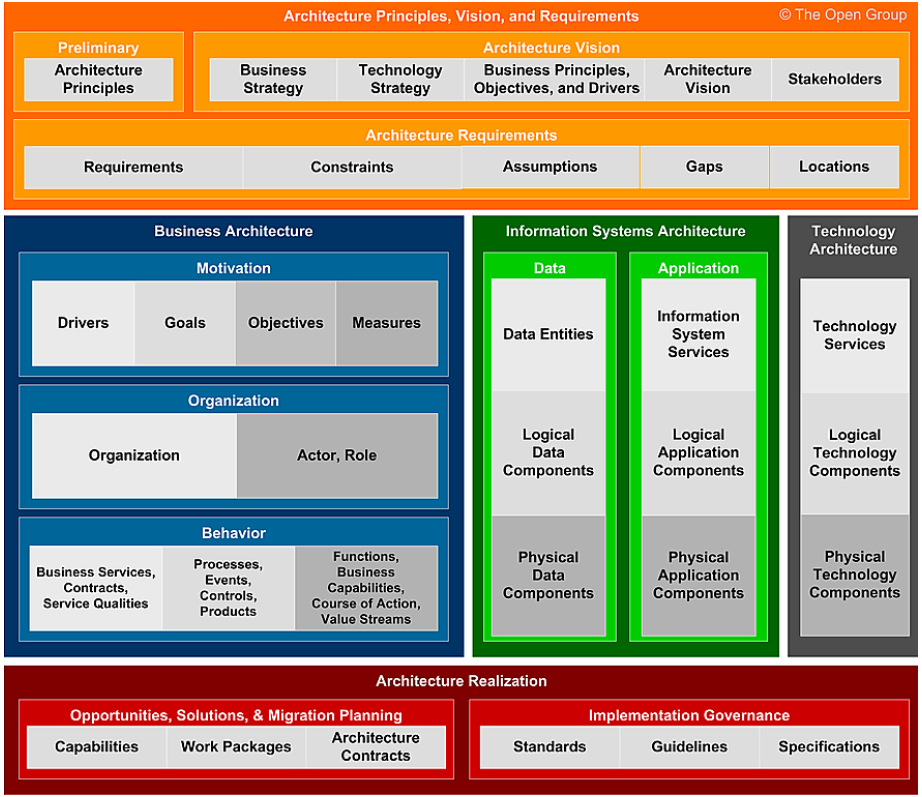


Defines the organizational structures, processes, roles, responsibilities, and skills/knowledge needed to realize a mature architectural capability (ensuring business alignment)

Source: https://pubs.opengroup.org/architecture/togaf9-doc/m/chap39.html

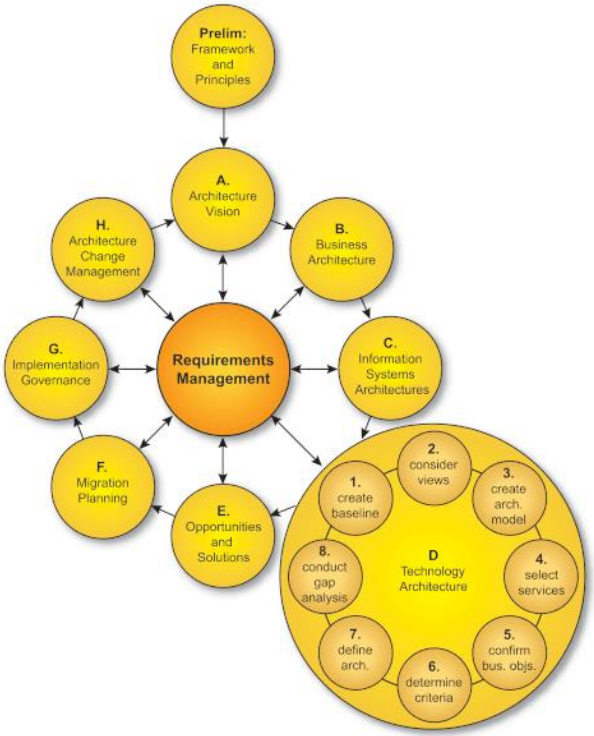
Architecture Content Framework

Provides a structured taxonomy for the types of architectural artifacts and content that can prove useful to an enterprise when building out an architectural capability



Architecture Development Method (ADM)

- Describes an iterative methodology for defining and building an enterprise architecture capability
- Reflects cyclical nature of continuous feedback and improvement
- Includes ongoing assessment of progress and activity at each step against defined requirements
- Each step further outlines a set of sub-steps that inform specific details of an activity
- Structured and meant to be adapted to your organization



Use Case – Methodology

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.

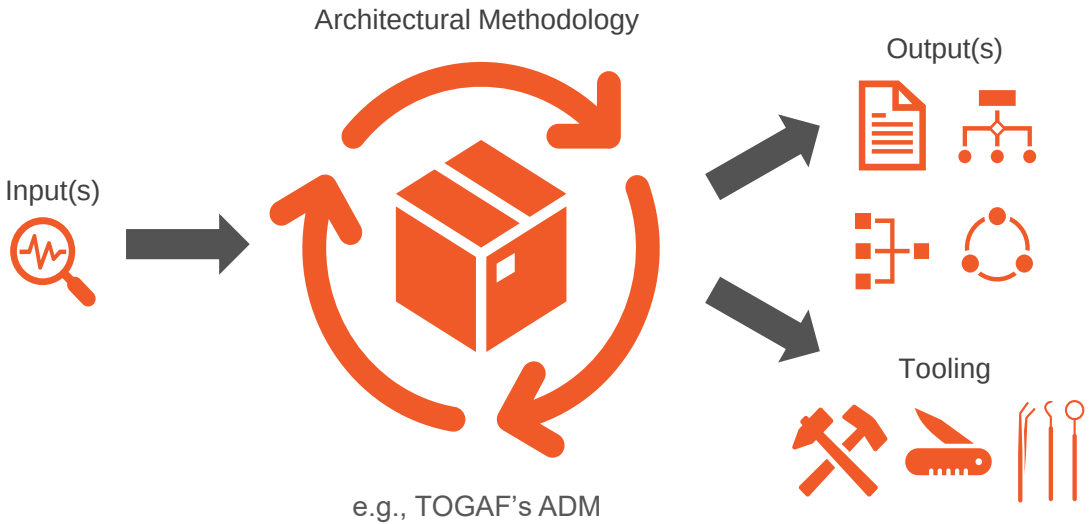
Also, the business would like to be able to leverage the different dimensions of data housed within this system for **Machine Learning** purposes but, at present, the data is difficult to extract and utilize. The current mainframe system supports **external customer access through a Web portal**, **internal 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:

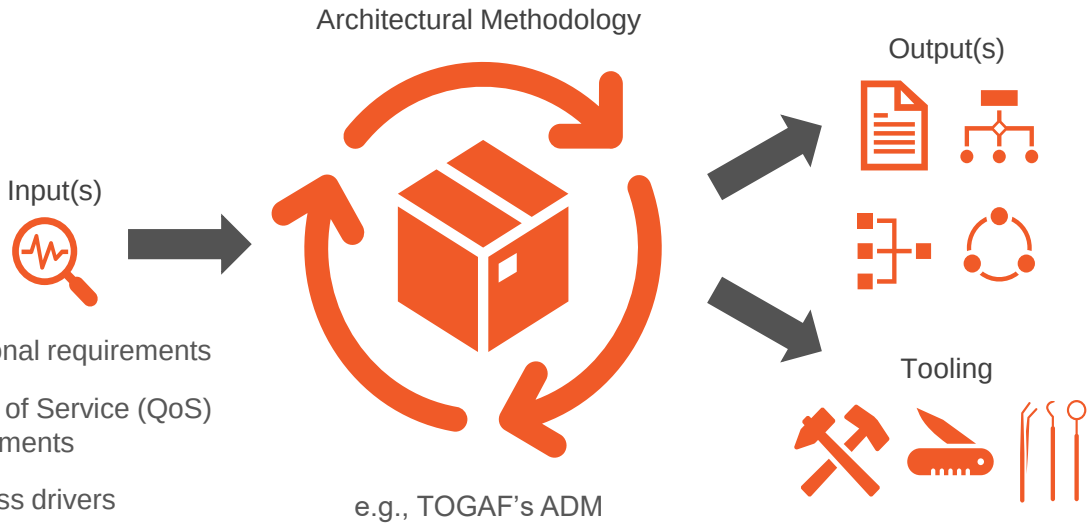
- What do you see as the business, technology, data, and application architecture concerns?

Outputs & Tooling

Architectural Artifacts

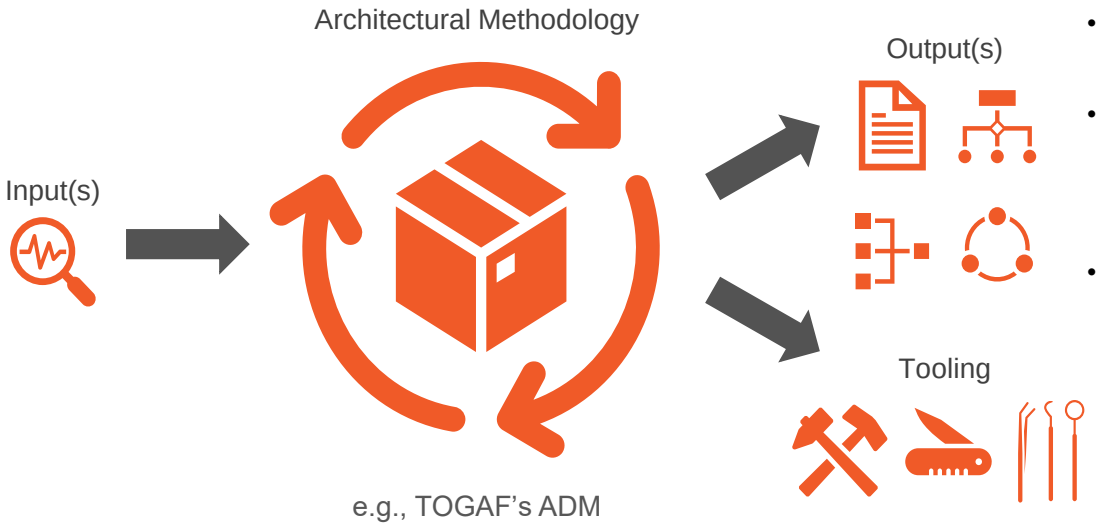


Architectural Artifacts



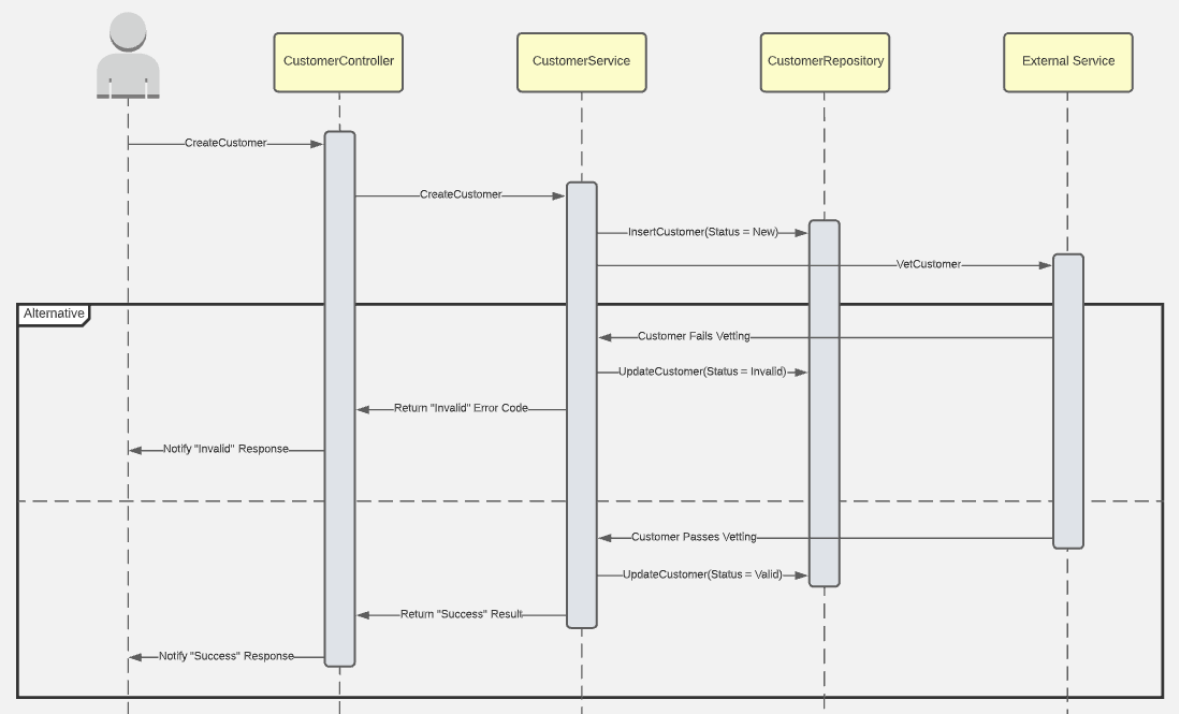
- Functional requirements
- Quality of Service (QoS) requirements
- Business drivers
- Technical drivers
- Organizational drivers

Architectural Artifacts

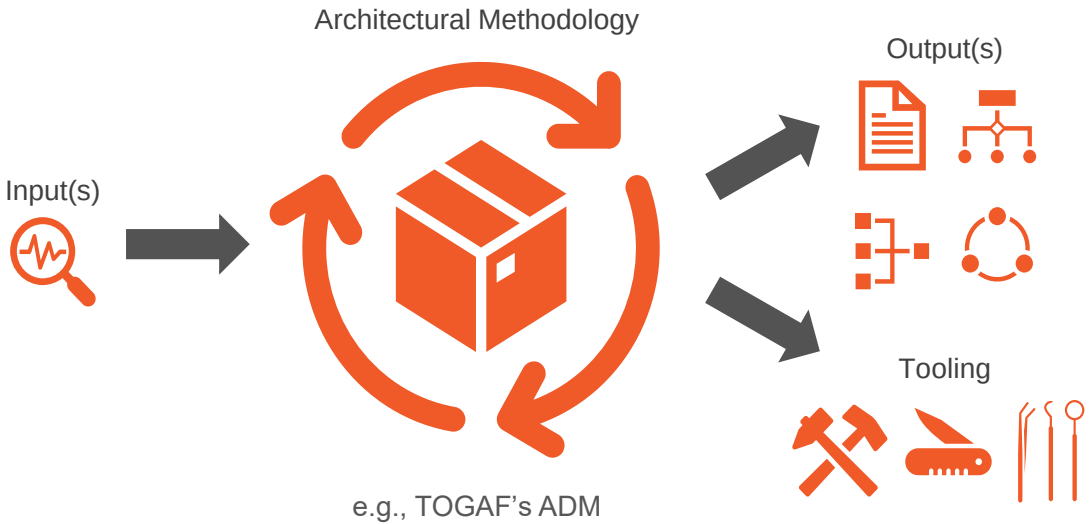


- Documentation of standards & best practices
- Diagrams (e.g., sequence diagrams, logical/physical architecture diagrams, ERDs)
- Processes that enable consistent application of the capability of architecture

Sample Sequence Diagram

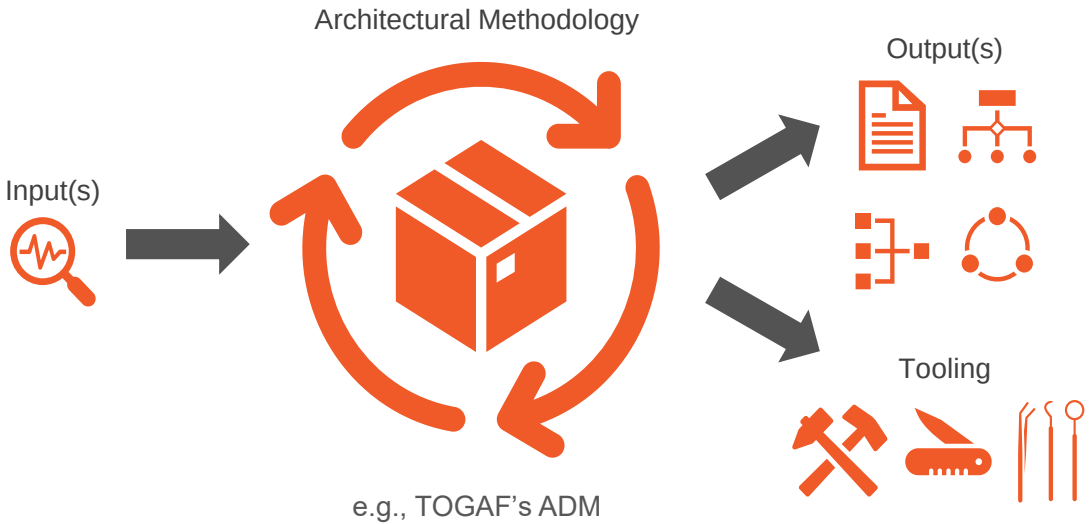


Architectural Artifacts



- 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?

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

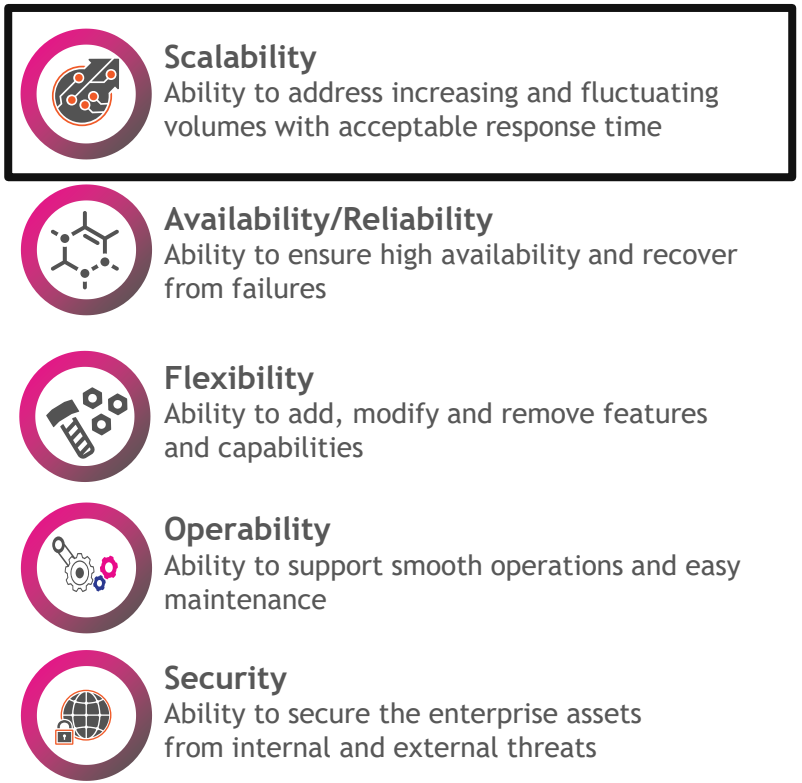
Operability
Ability to support smooth operations and easy maintenance

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

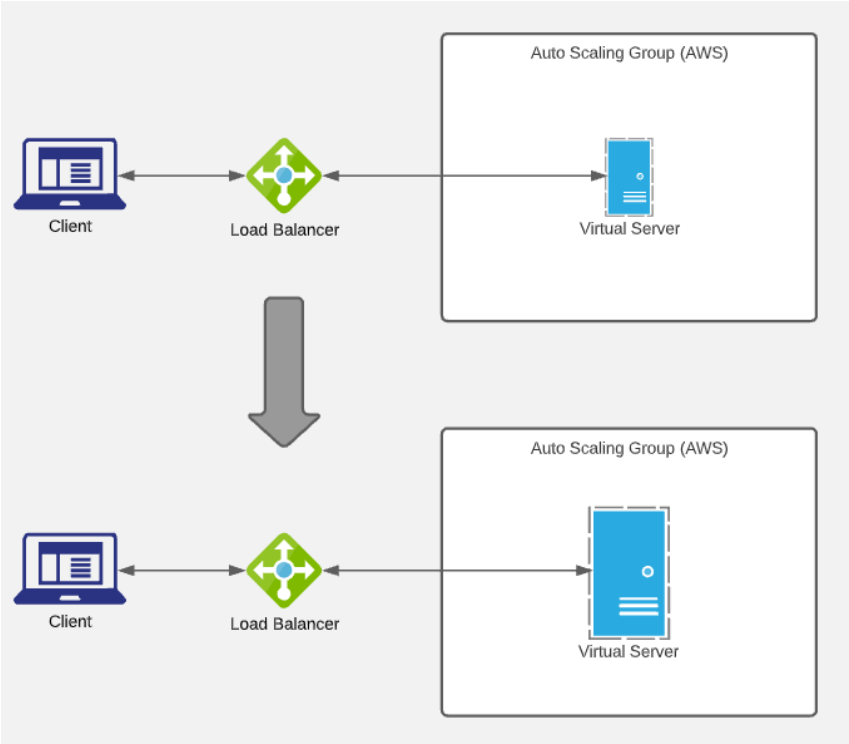
Quality attributes as Architecture Requirements

- 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

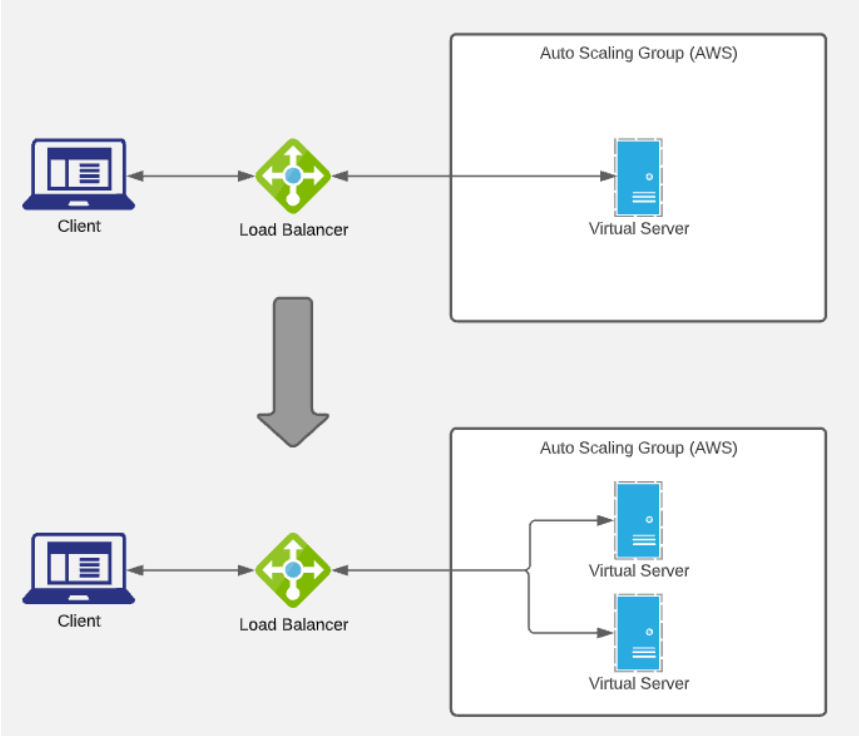
Five fundamental quality attributes



Vertical Scaling (Scaling “Up”)



Horizontal Scaling (Scaling “Out”)



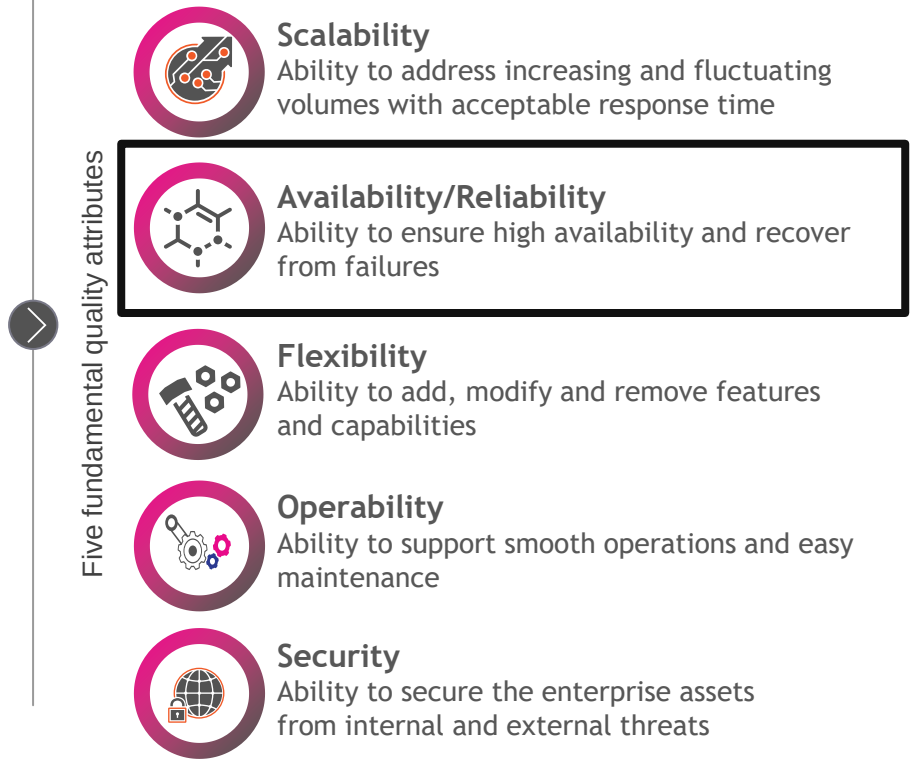
Moment of Reflection

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

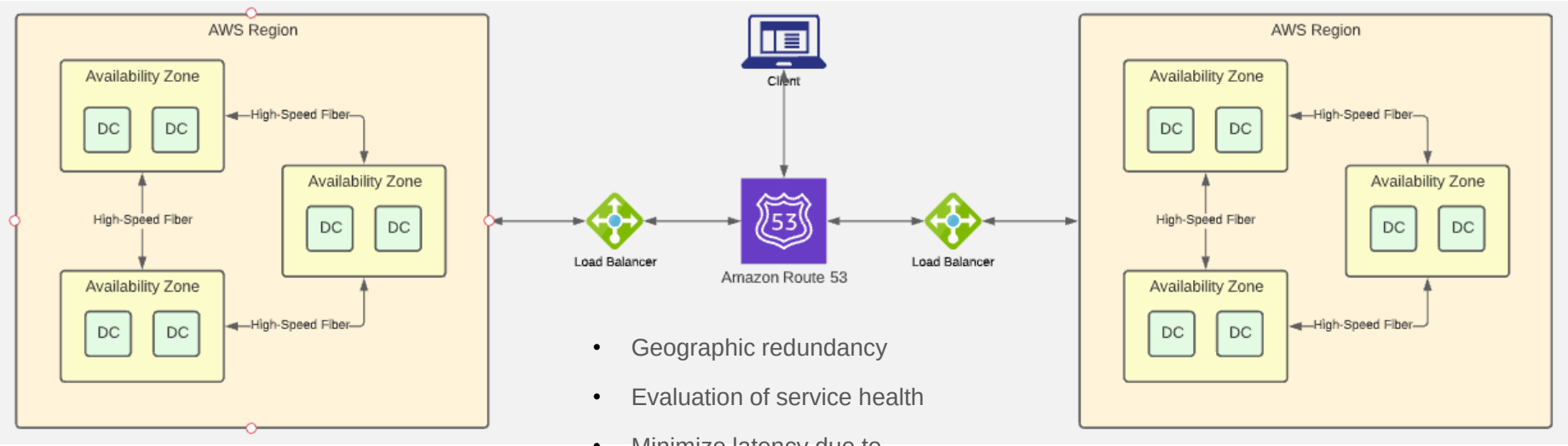
Type your answer in chat or come off mute & share

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



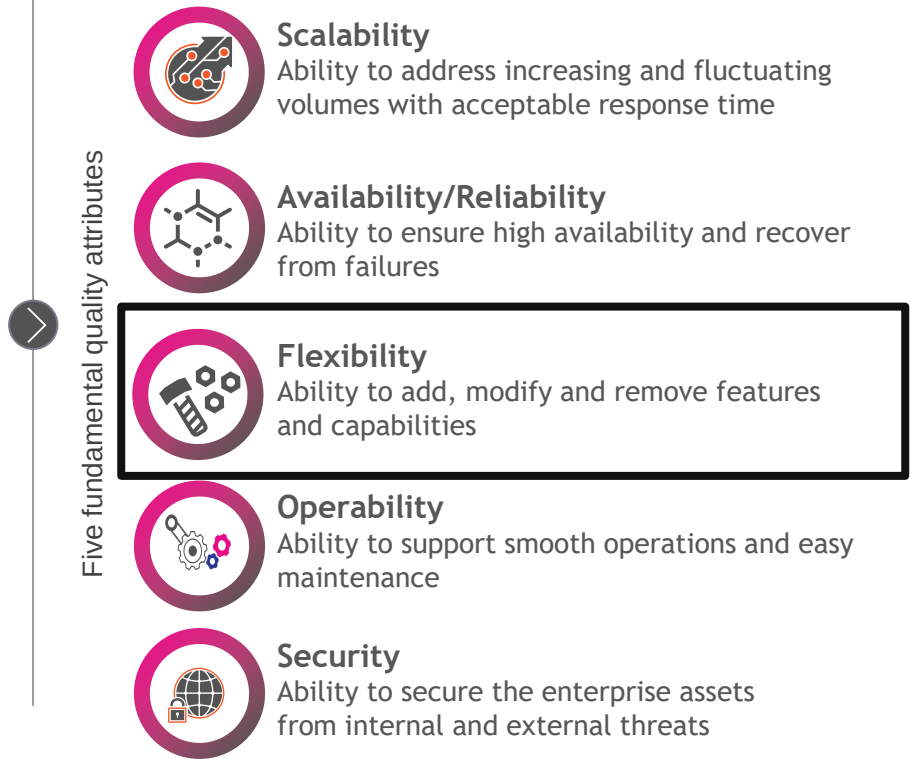
Availability/Reliability



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

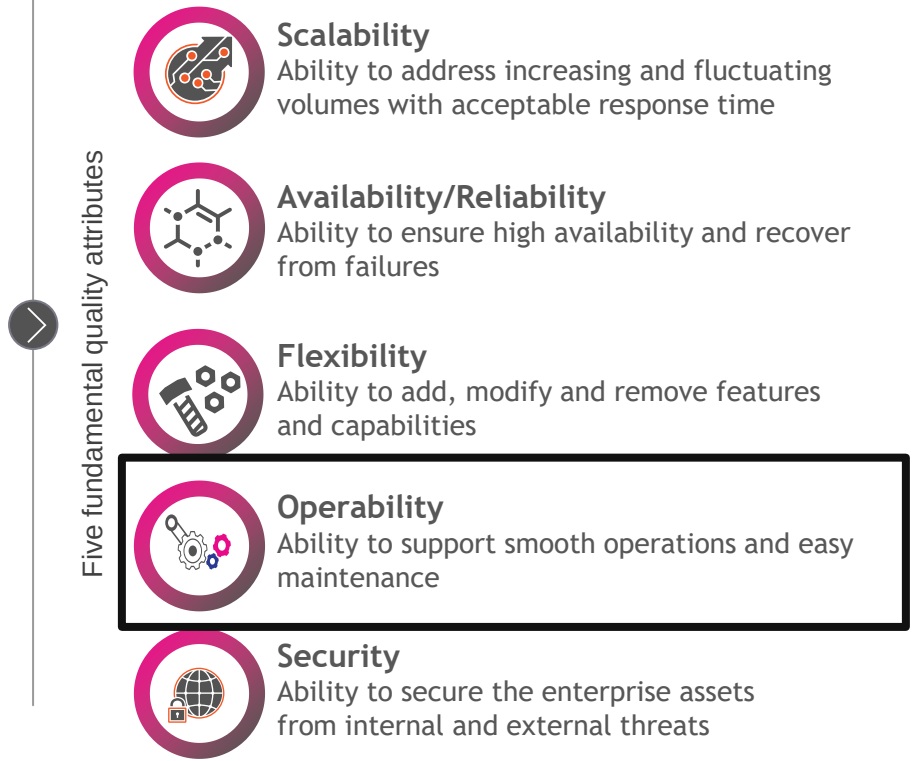
Quality attributes as Architecture Requirements

- 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

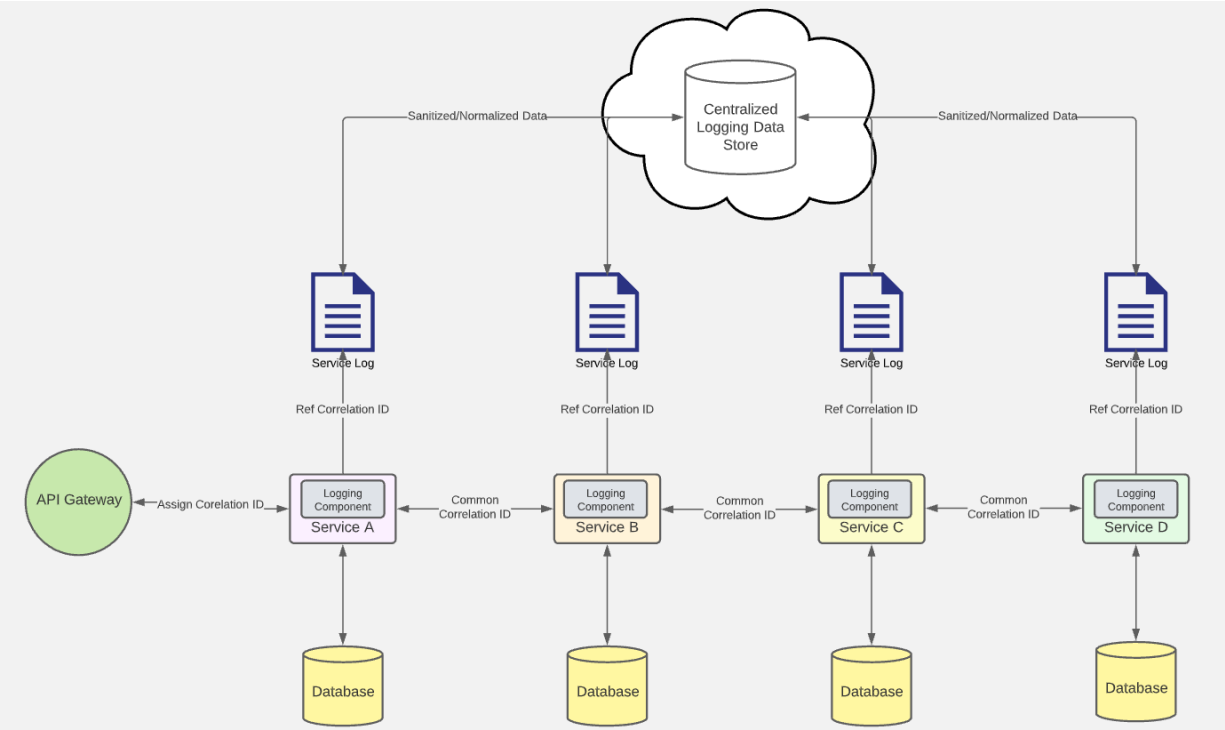


Quality attributes as Architecture Requirements

- 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.?”

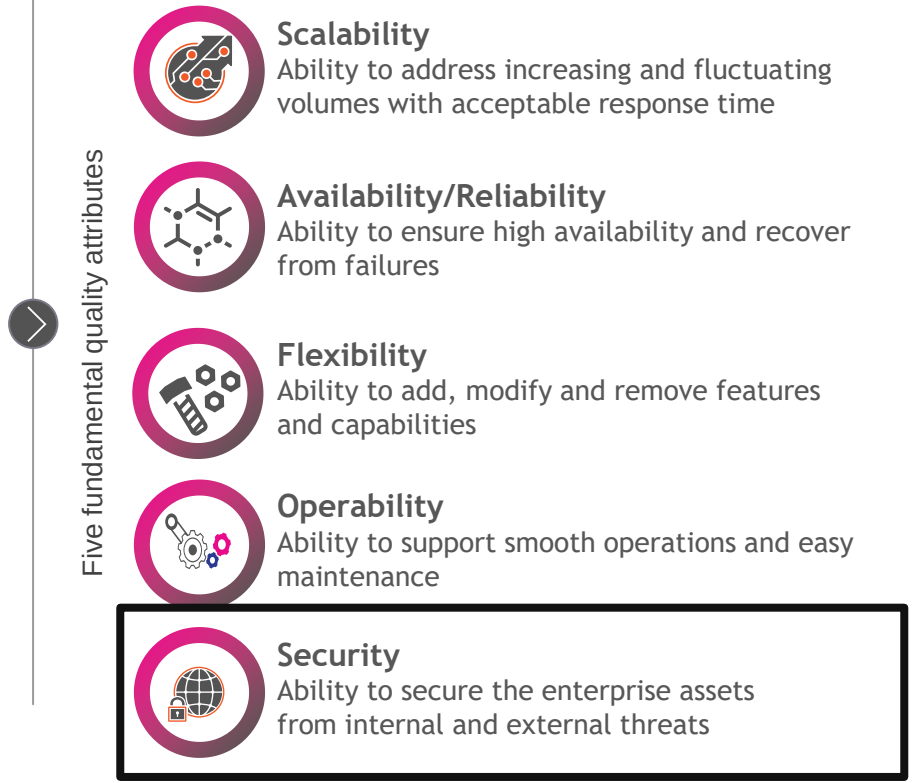


Operability

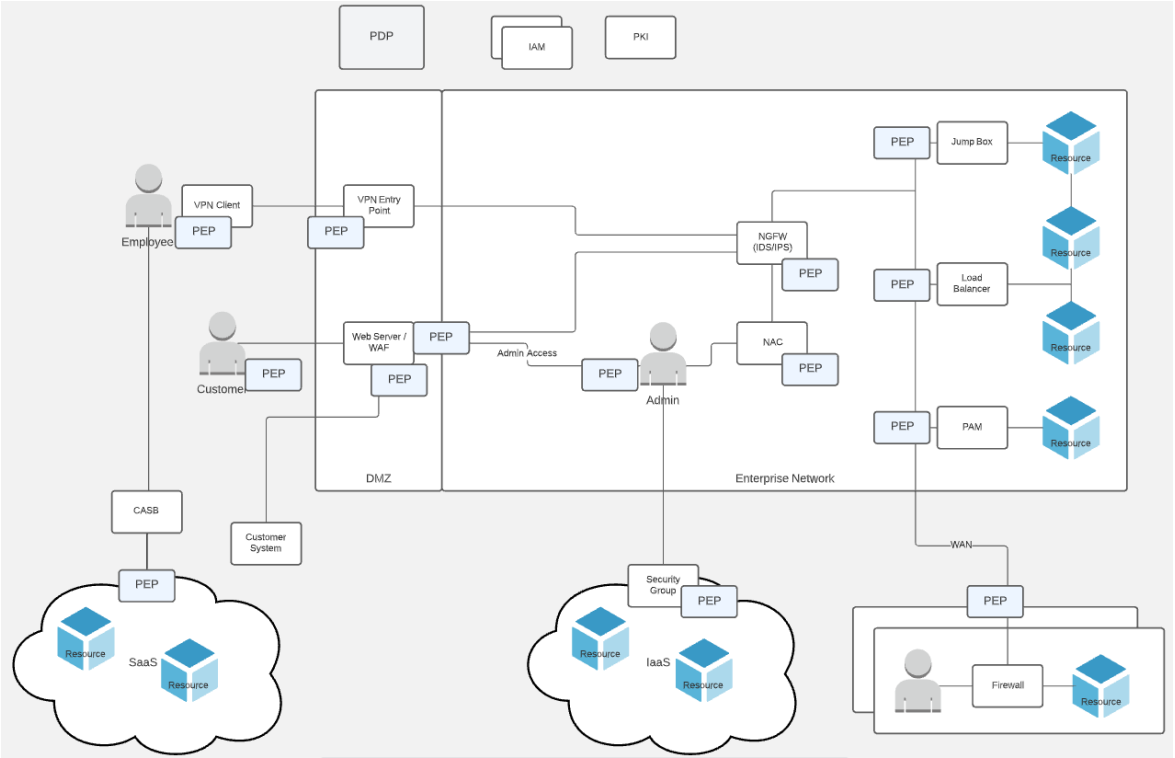


Quality attributes as Architecture Requirements

- 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.

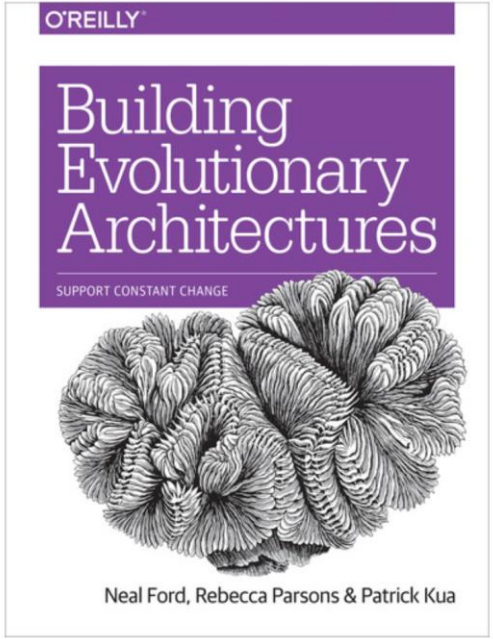
Also, the business would like to be able to leverage the different dimensions of data housed within this system for **Machine Learning** purposes but, at present, the data is difficult to extract and utilize. The current mainframe system supports **external customer access through a Web portal**, **internal 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)?

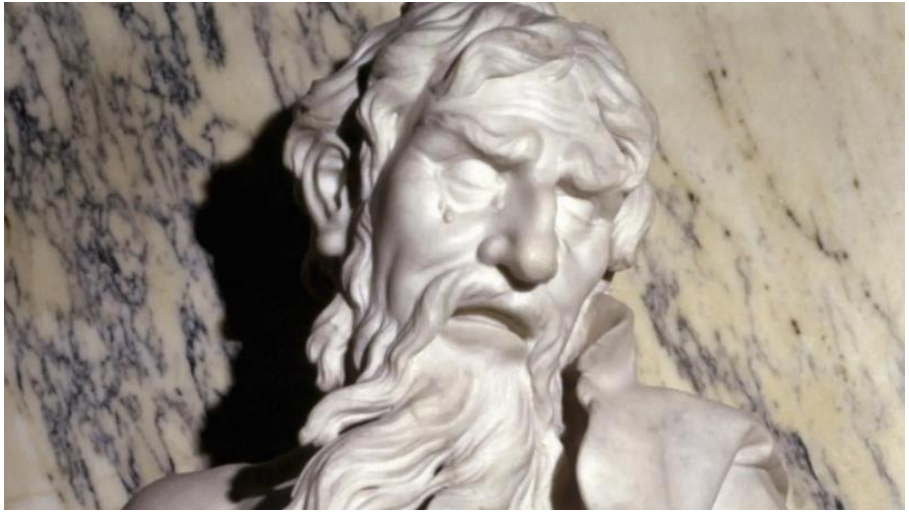
Evolutionary Architecture

Evolutionary Architecture



Great book on this topic!

Changes

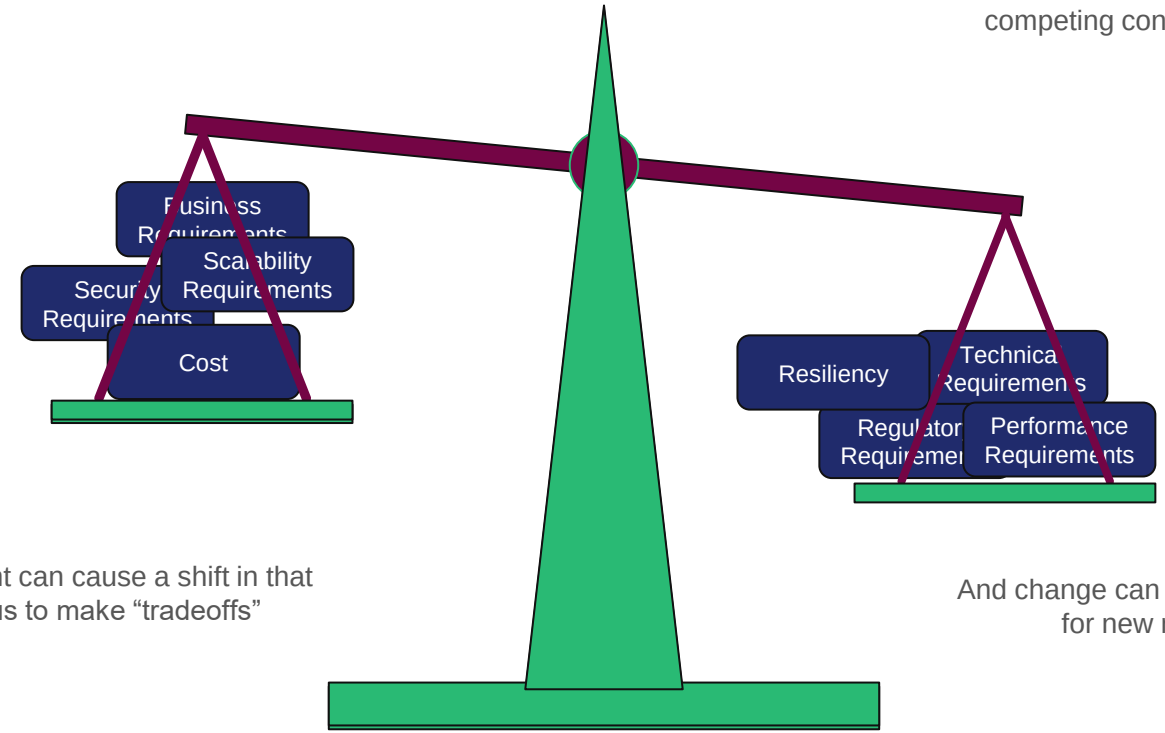


- Heraclitus (Greek philosopher) was quoted as saying “the only constant is change”
- When building software, we execute a project at a point in time
- However, the goal is for the software we build to provide value that outlasts the immediate
- For that to be a reality, we have to expect (and ideally architect for) changes
- Change comes in many forms

Change due to business innovation, mergers/acquisitions, a new regulatory requirement, evolving security threats, advancements in technology, etc.

Architectural Balance

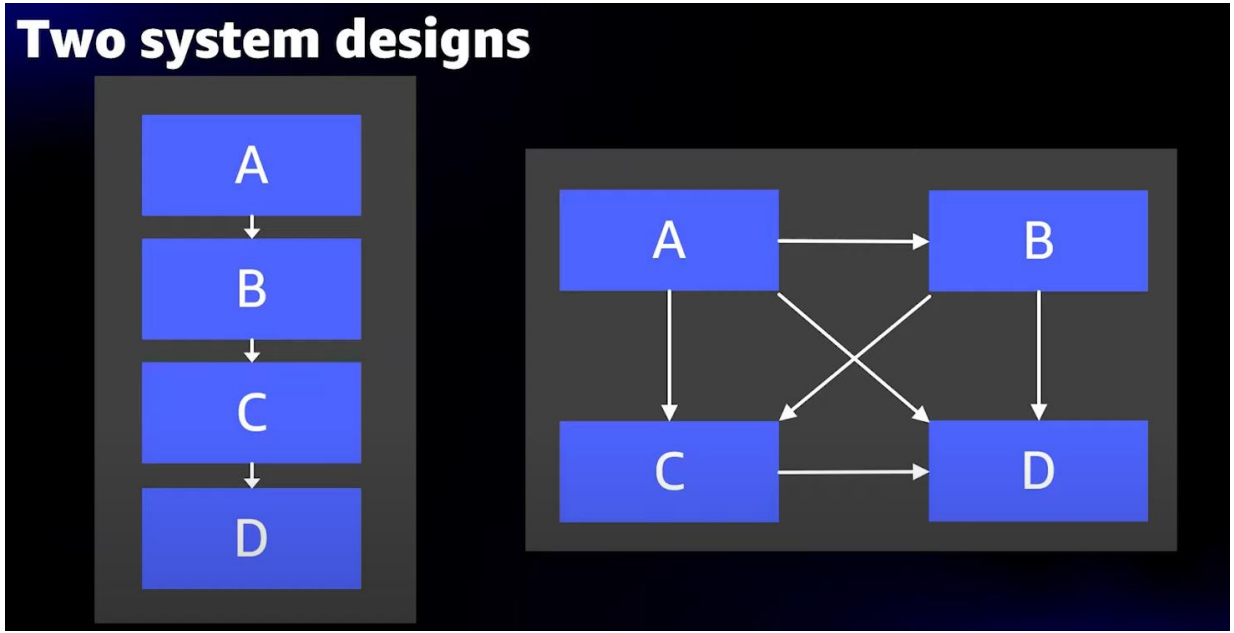
As architects, we are forced to balance multiple competing concerns



Each new requirement can cause a shift in that balance, forcing us to make “tradeoffs”

And change can be a common source for new requirements

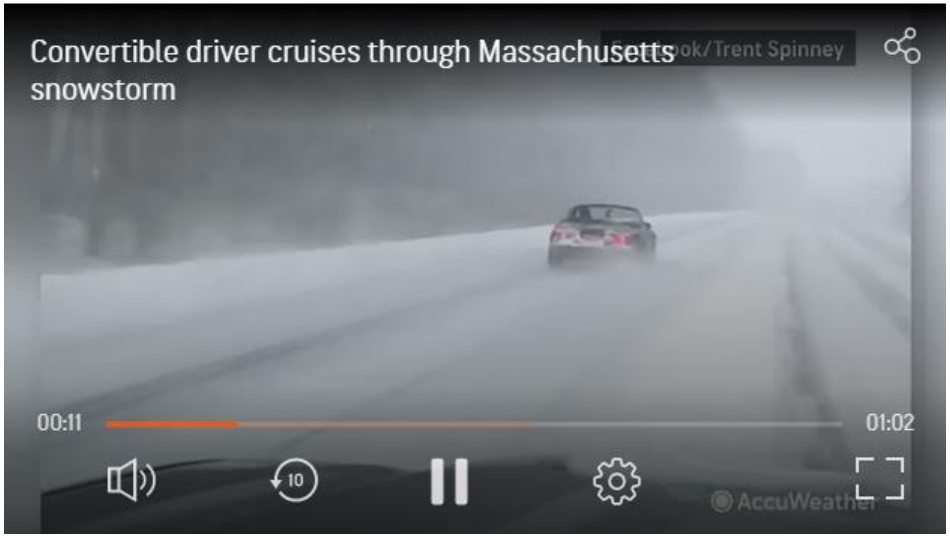
Balancing Competing Architectural Concerns



Given these two options for designing a distributed system, what are the architectural tradeoffs for each? How would you balance the architectural concerns associated to each?

Architectural Awareness

Balancing competing architectural concerns requires that we maintain awareness of our surroundings



We cannot bury our heads in the sand (or snow); otherwise, we run the risk of missing critical details that can mean the difference between success and suboptimal

Source: Accuweather.com

We must be able to see both the  and the  s

Two Types of Change

Incremental

How do we move our architectures forward in small but meaningful steps, balancing the shifts in equilibrium that can come with each change?

Guided

What are the most important architectural characteristics (or dimensions) of our designs, how do we define progress, and how do we assess current state against target state?

Incremental Change

- Can involve both how we build software and how we deploy it
- With incremental steps, smaller scope of change
- What is “far enough” but “not too far”?
- From a deployment perspective, can potentially run multiple versions side-by-side (e.g., for a microservice), allowing callers to gradually migrate from old to new
- When old is no longer used, can be deadfiled

The challenge here can be ensuring our technology landscape remains streamlined and as “uncluttered” as possible

Guided Change

Prioritize the most important architectural characteristics of the design

Identify a target state for the system

Leverage objective mechanisms to assess progress toward target state and adherence to those prioritized characteristics

If we feel our system veering off-course from the defined objectives, course correct

This assessment coupled with the incremental nature of change can help prevent more significant issues

Architectural Dimensions

- Includes the multiple “-ilities” typically associated with QoS requirements
- But can include other architectural concerns as well
- For example:

Technical

- Frameworks, libraries, languages

Data

- Schemas, table design, indexing, performance tuning

Security

- Policies, guidelines, tooling, data protection

Operational/System

- How the architecture maps to existing infrastructure

4+1 Architectural View Model

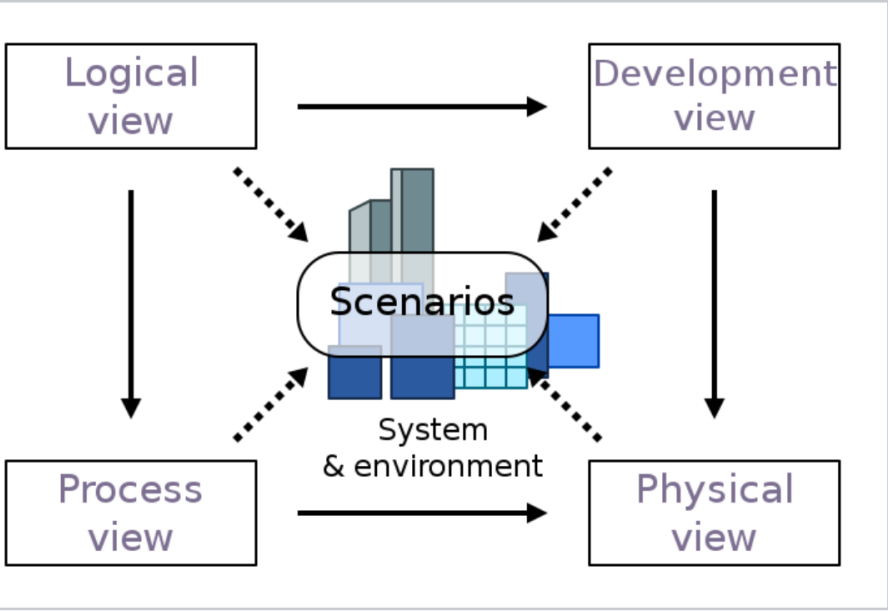


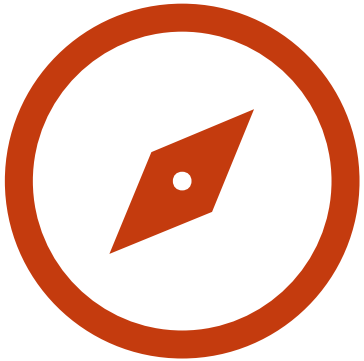
Illustration of the 4+1 Architectural View Model.

- Logical view – the functionality provided by the system to end-users
- Process view – the dynamic, runtime behaviour of the system; the system workflow
- Development view – the programmer's perspective; AKA the implementation view
- Physical view – the deployment view; what does the system look like running on production servers and production networks
- Scenarios – use cases that provide validation and inform the architecture

Source: https://en.wikipedia.org/wiki/4%2B1_architectural_view_model

Fitness Functions

- As part of guided change, fitness functions are those objective measures we can use to assess and evaluate progress
- Multiple types can be leveraged for different prioritized architectural dimensions
- Methodology of execution can vary as well depending on the dimension in question
- Ideally, provide a “quality gate” that can be used to identify issues early and prevent sub-optimal results



Fitness Functions

Atomic

- Exercise and verify a specific unit of architectural quality
- For example, a unit test that automatically validates a key architectural characteristic

Holistic

- Exercise a combination of architectural concerns
- For example, performance & scalability
- Ensure combinations of architectural concerns “play well” together

Fitness Functions

Triggered

- Run based on the occurrence of a specific event
- For example, a build pipeline that ensures passing tests before deployment

Continual

- Run continually to assess architectural quality over time
- For example, average transaction speed traced across a series of requests

Fitness Functions

Static

- Binary pass/fail

Dynamic

- Nuanced evaluation
- For example, accounting for some acceptable level of performance hit as the system scales to handle a large volume of requests

Fitness Functions

Intentional

- What's known

Emergent

- What's learned as design and development of the system proceeds

Fitness Functions

Examples

- Integrating combination of all unit tests passing with minimum coverage percentages into CI/CD pipeline, failing build if target measure unmet
- 0 critical or high security vulnerabilities found from static scans, dynamic scans, or open source component scans
- Successful adherence to RTO & RPO during disaster recovery exercises
- Adherence of average response times as measured against target SLAs under normal load (50^{th}, 90^{th}, 95^{th} percentiles)
- Adherence of average response times as measured against target SLAs under stress (50^{th},90^{th}, 95^{th} percentiles)

Use Case – Fitness Functions

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.

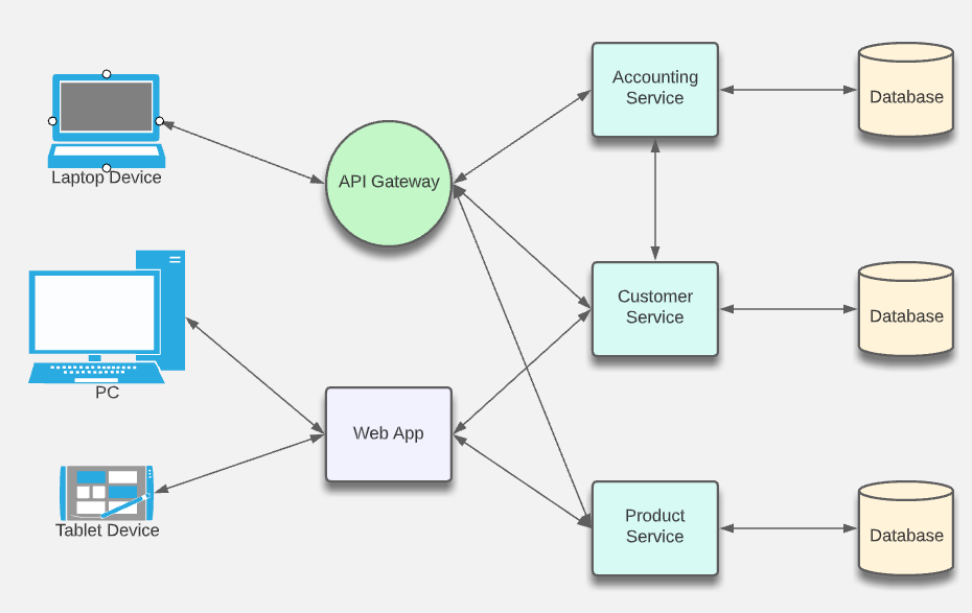
Also, the business would like to be able to leverage the different dimensions of data housed within this system for **Machine Learning** purposes but, at present, the data is difficult to extract and utilize. The current mainframe system supports **external customer access through a Web portal**, **internal 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:

- Choose 1 of the Quality of Service requirements discussed previously. Can you think of a specific fitness function that could be used to help guide & validate the evolution of the architecture over time relative to that QoS requirement?
- Where (if anywhere) might you be able to apply incremental change in the execution of this overall effort?

Modern Architectural Styles & Patterns

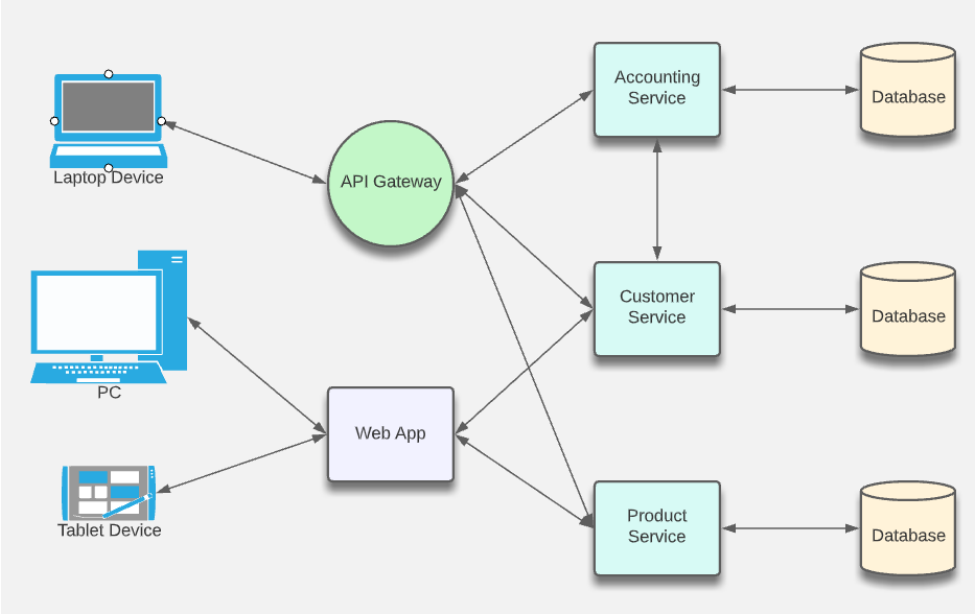
Microservices



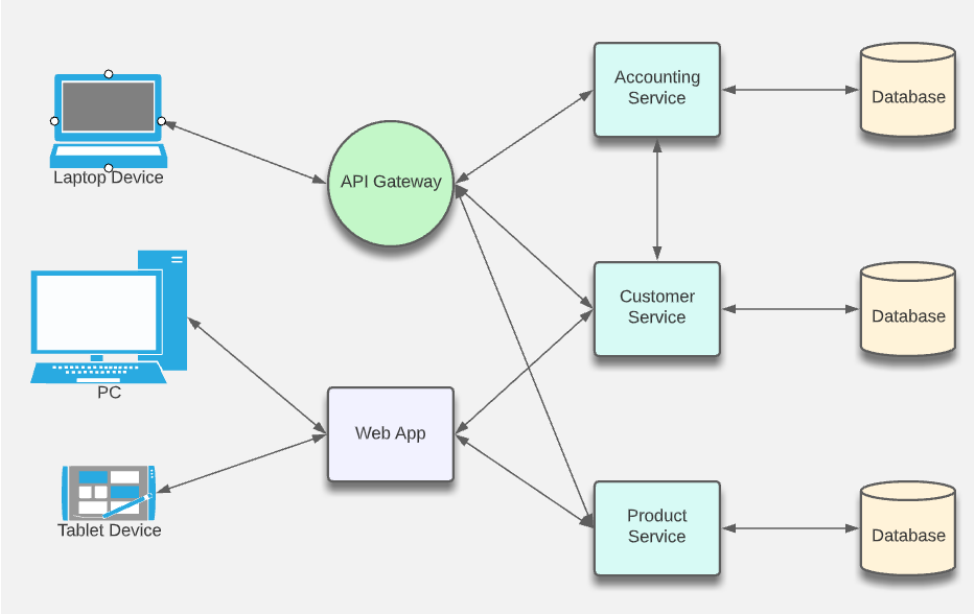
Microservices

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



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

Technology Options for Building APIs

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

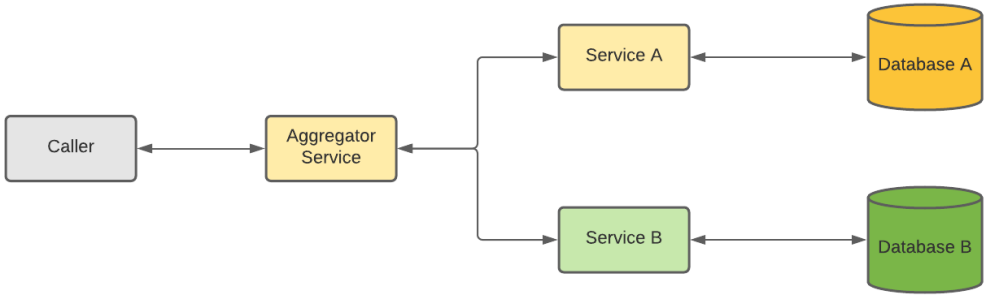
- ✓ Implementation using a serverless platform like AWS Lambda
- ✓ Exposing via Robotic Process Automation (RPA)
- ✓ Exposing as a set of endpoints using the Mulesoft platform (or similar)
- ✓ Implementation using a microservices-based approach & architecture

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)

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:

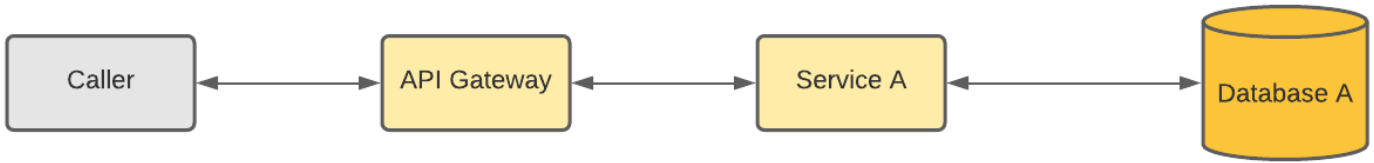
- Helps you practice DRY (Don’t Repeat Yourself)
- Supports logic at time of aggregation for additional processing

Potential “gotchas”:

- Performance if downstream microservices not called asynchronously

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:

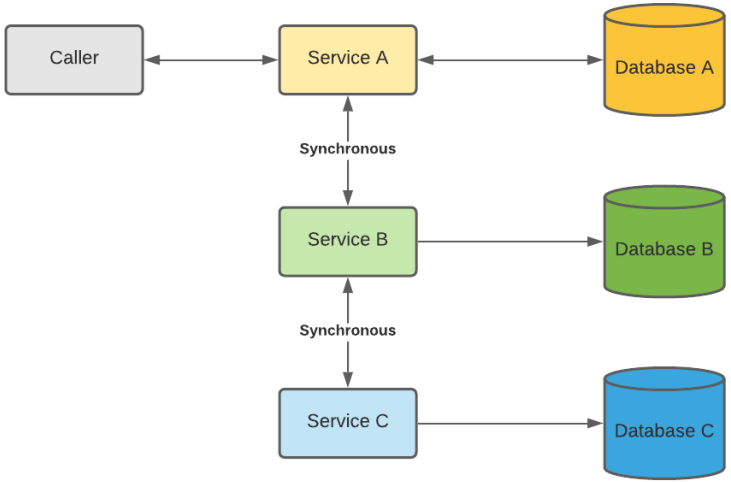
- Usually, built in throttling (to protect against DDoS)
- Can support translation between caller and downstream microservice for protocol, message format, etc.
- Provides a centralized point of entry for AuthN/AuthZ
- Can be combined with load balancing to enable scalability and resiliency
- Centralized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

Potential “gotchas”:

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

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:

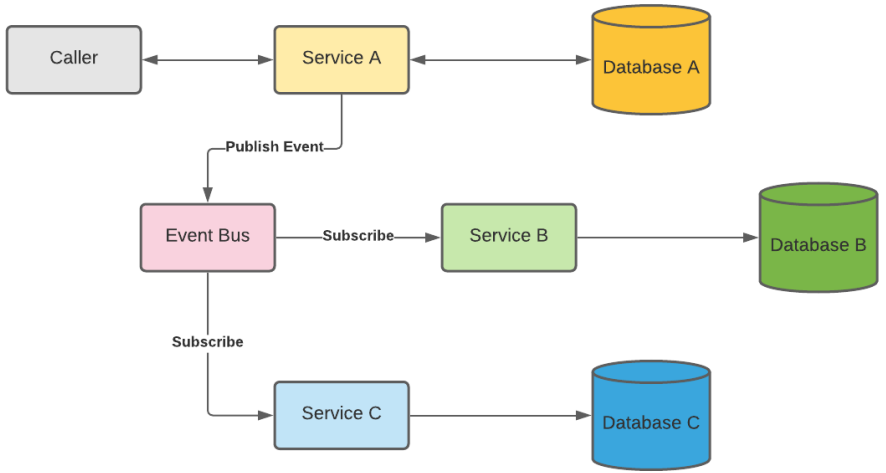
- Supports composition of microservices in a workflow that must happen in sequence
- Synchronous communication typically less complex

Potential “gotchas”:

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

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:

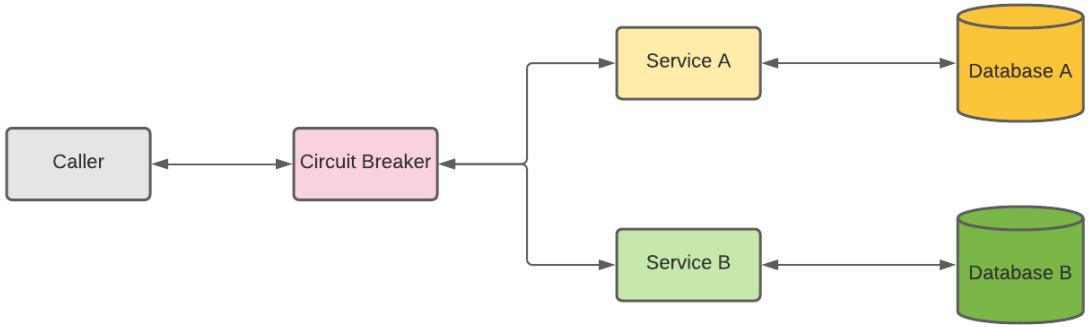
- Promotes looser coupling in cases where synchronous calls are not required
- Allows multiple services the option of being notified (vs. point-to-point)
- If given consumer is down, producer can continue to send messages and they won't be lost

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

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:

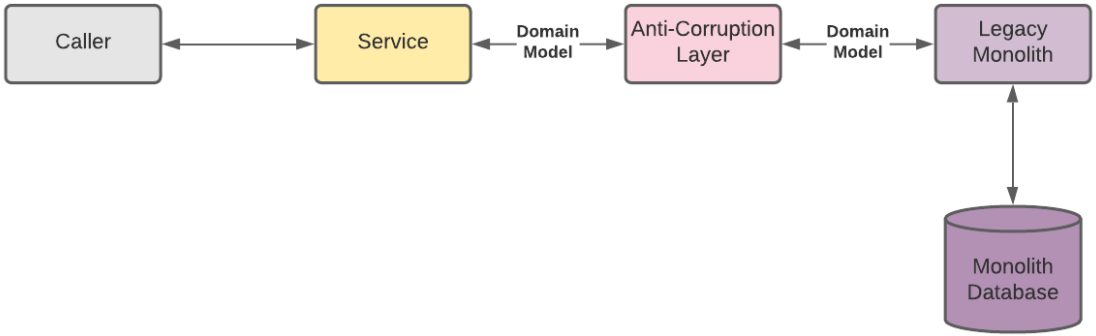
- Helps to optimize network traffic by preventing calls that are going to fail
- Can help prevent “noise” and network overload

Potential “gotchas”:

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

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:

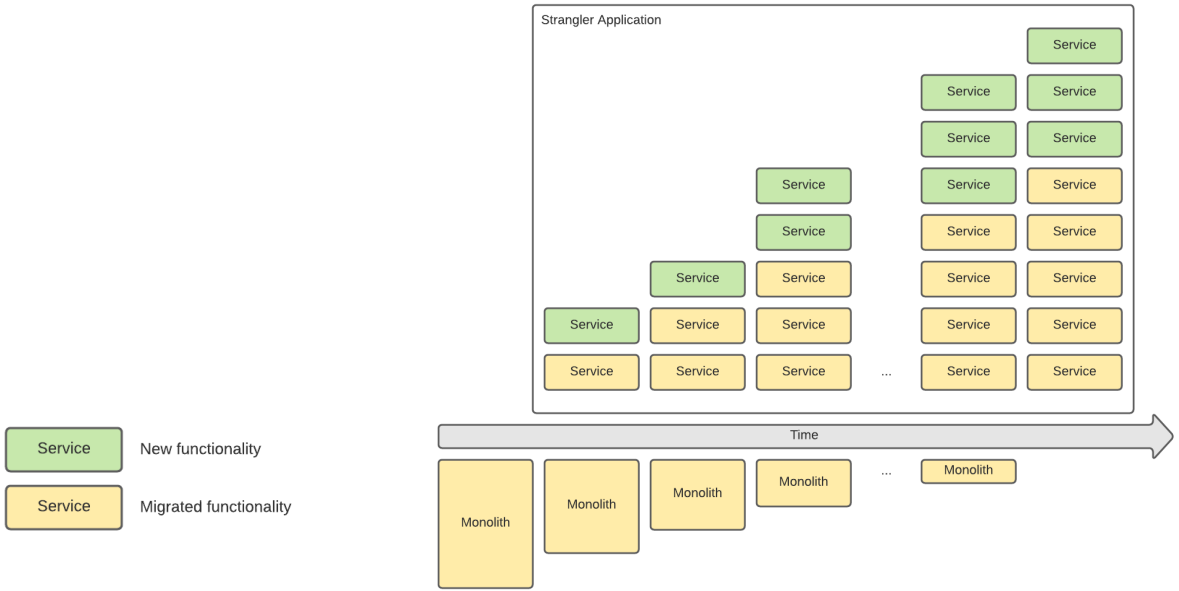
- Prevents crossing of boundaries and “leaking” of sub-optimal logic
- Helps to keep this insulation transparent between communicating parties

Potential “gotchas”:

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

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”:

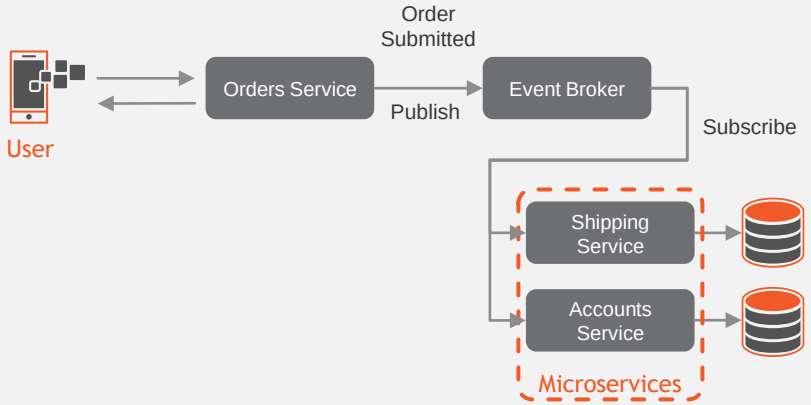
- Requires business to be able to absorb gradual
- Adds complexity – coordinating new functionality, migrated functionality, and proper integration between them
- While transition in progress, will have to maintain two paths

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

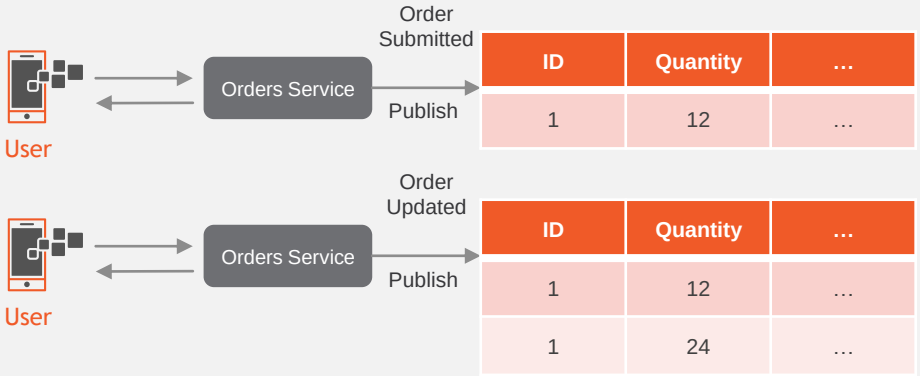


Event-Driven Architectures

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

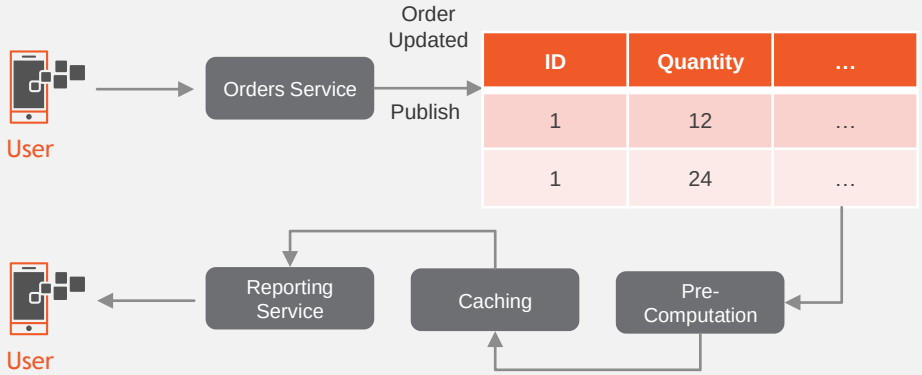


Event-Driven Architectures

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-Driven Design (DDD)

Domain-Driven Design (DDD)

- Process of designing systems aligned to business domain
- Drives business language into the technology – engineer in terms of business process & impact
- Can enable IT to be an enabler and even an accelerator (vs. another “cost center”)
- Helps with management of types of coupling
- Direct correlation can be drawn between domains and microservices

Defining Boundaries

- Enforcing strong boundaries and controlled interfaces can help mitigate coupling
- Domains become realms of cohesion
- Controlled integration – i.e., what do I allow others to “see”?
- DDD is about helping us determine where the boundaries should exist

Some Key Terms

Bounded Context

- Provides a place for a set of terms to have a specific business meaning
- Contexts allow for variance in technology from domain-to-domain
- An entity may exist with the same name in two different bounded contexts, but its context will dictate its meaning

Some Key Terms

- Ubiquitous Language
  - Shared language between tech and business
  - Using language that models the business
  - Promotes collaboration between domain experts and tech

Some Key Terms

Domain

- A “slice” of your software system
- Ideally, maps to a bounded context (and potentially one or more microservices)
- Can encapsulate subdomains
- Three major types: Generic, Supporting, and Core

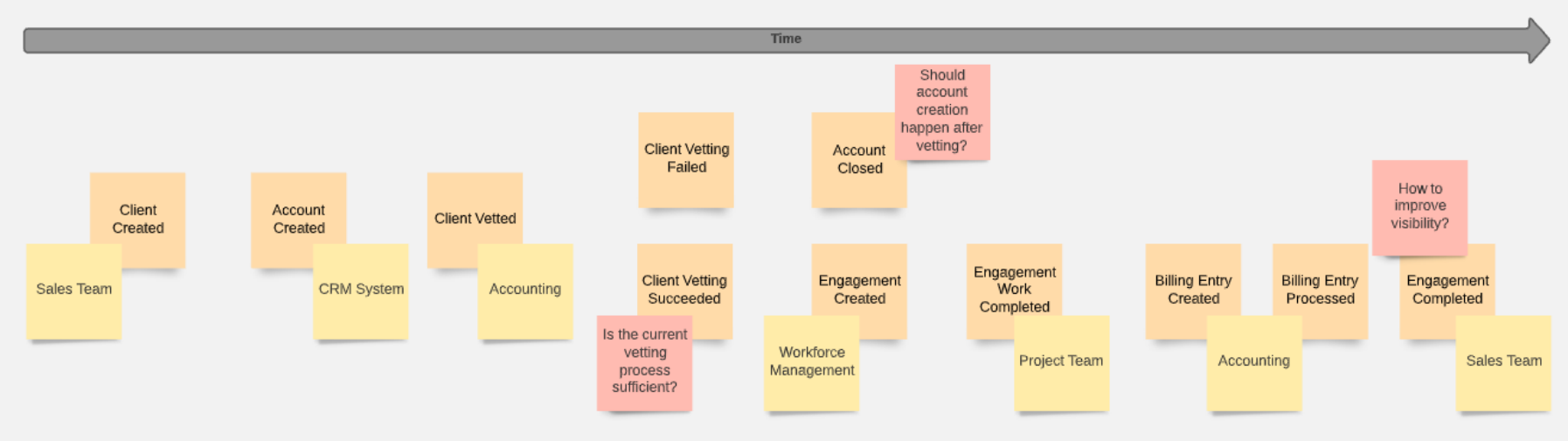
Some Key Terms

Generic Domain – “Buy vs. Build”, SaaS, use 3rd party or framework

Supporting Domain – Develop, maybe outsource

Core – Unique and distinguishing; where lion’s share of efforts are focused

EventStorming - Example



DDD – Cheat Sheet

Strategic

- Analyze Business Domain
- Identify Subdomains
  Core, Supporting, Generic
- Domain Modeling
- Ubiquitous Language
- Defining & Integrating
  Bounded Contexts

Tactical

- Layered Architectures
- Defining Services
  Domain & Application
- Defining Aggregates
- Defining Entities & Value Objects
- Defining Domain Events

Use Case – Architectural Styles & Patterns

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).

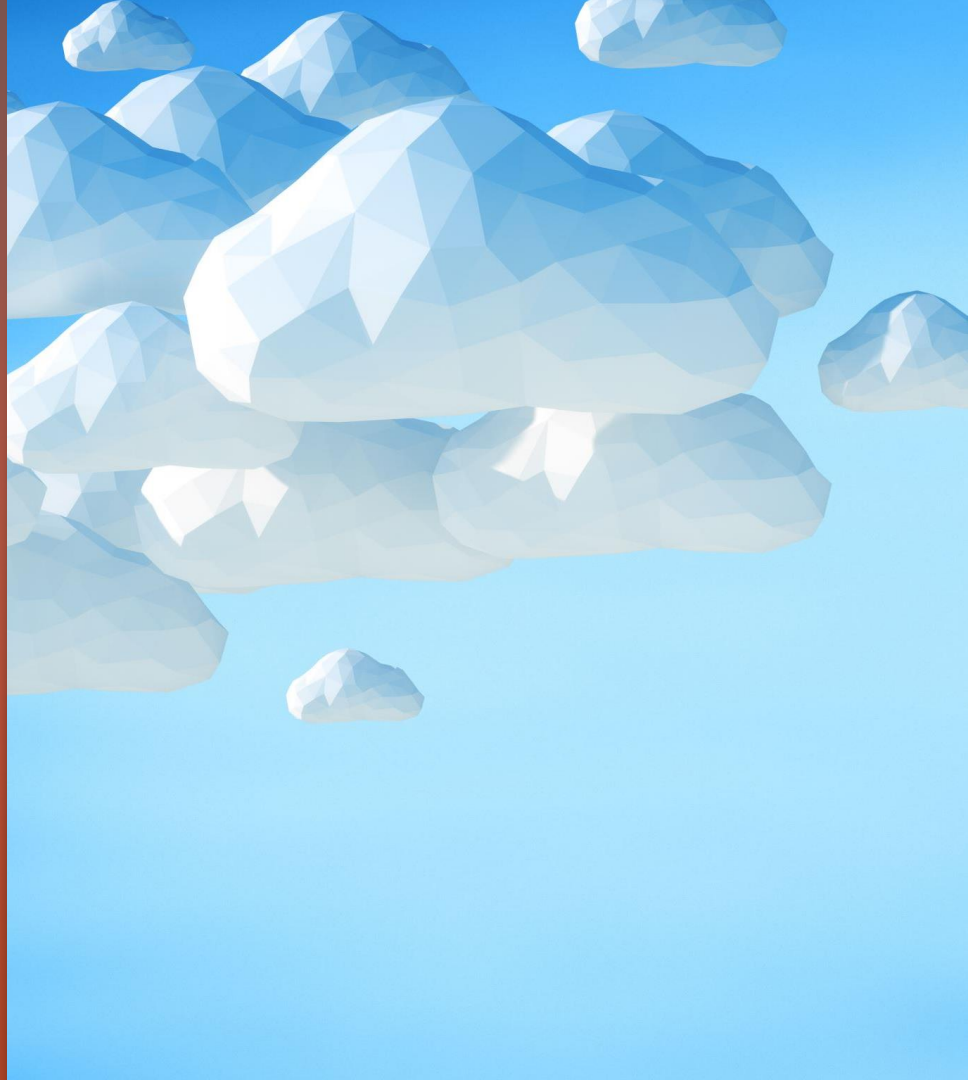
Currently, the application leverages a mainframe-based relational data store to store and serve all data within the large and tightly-coupled workflow. There are several legacy technologies in play and the platform upon which the application is deployed is **several versions behind latest**. Key reports 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:

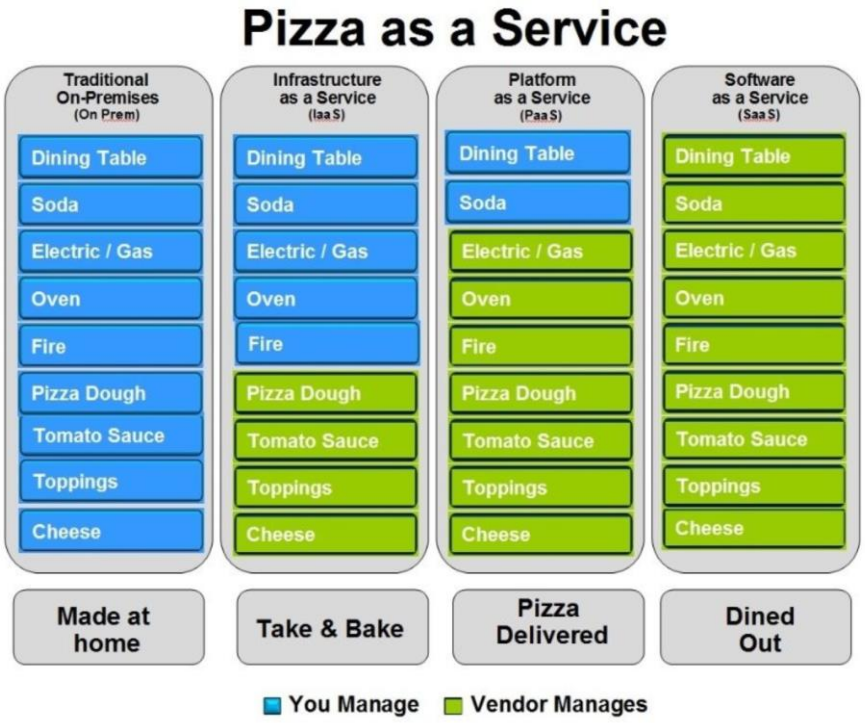
- First “blush”, where might you draw domain boundaries around the different areas of the system?
- Which of the architectural styles & patterns would you recommend employing in the design of this system?
- What are the advantages provided by each of your selected styles/patterns? What are some potential architectural challenges?

Architecting for the Cloud

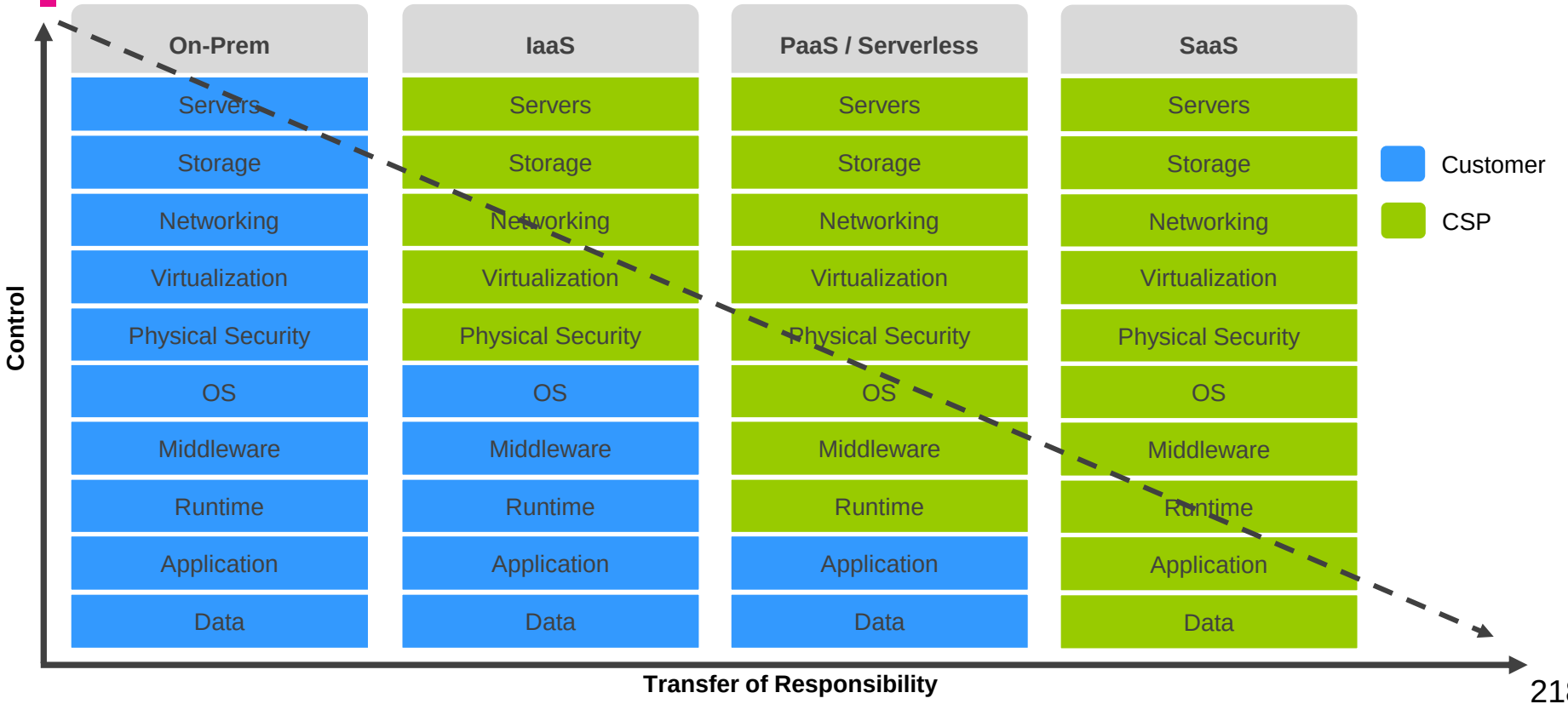


Pizza-as-a-Service

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

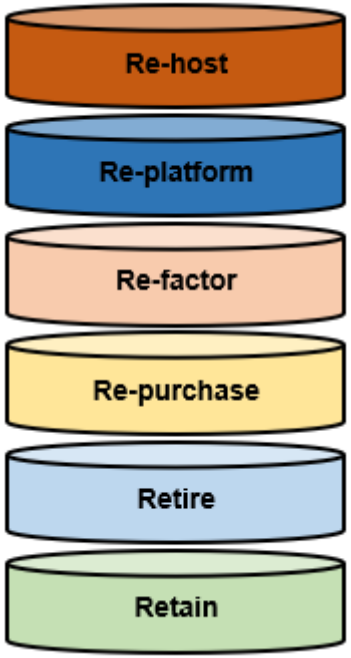


Side-by-Side Comparison



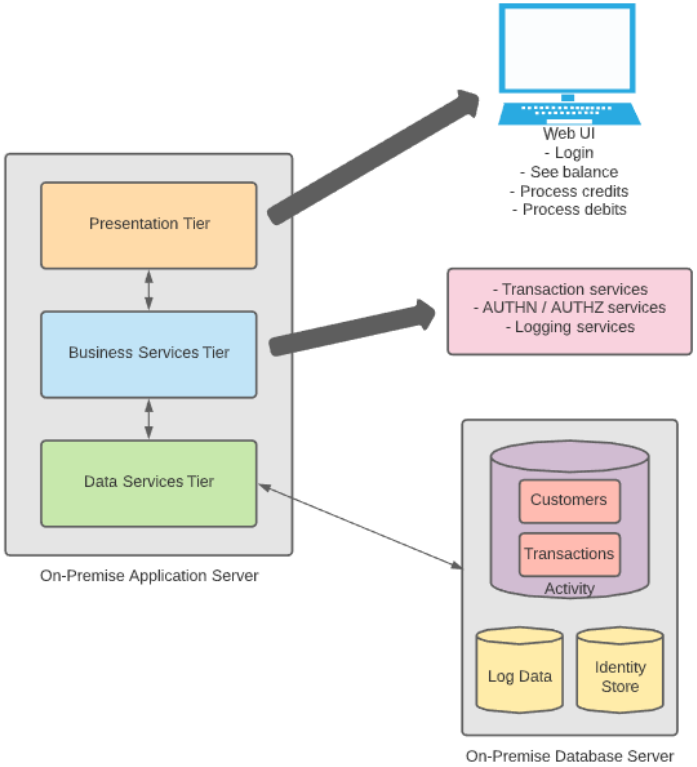
The 6 R's – Application Migration Strategies

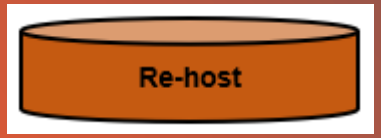
When moving from on-premise to the Cloud (or hybrid), the architect decides whether to migrate, modernize, or do some combination of the two



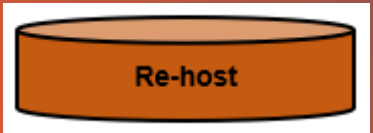
Source: https://aws.amazon.com/blogs/enterprise-strategy/6-strategies-for-migrating-applications-to-the-cloud/

Current State Architecture

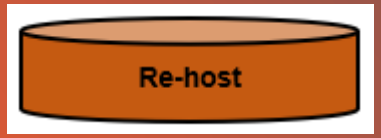


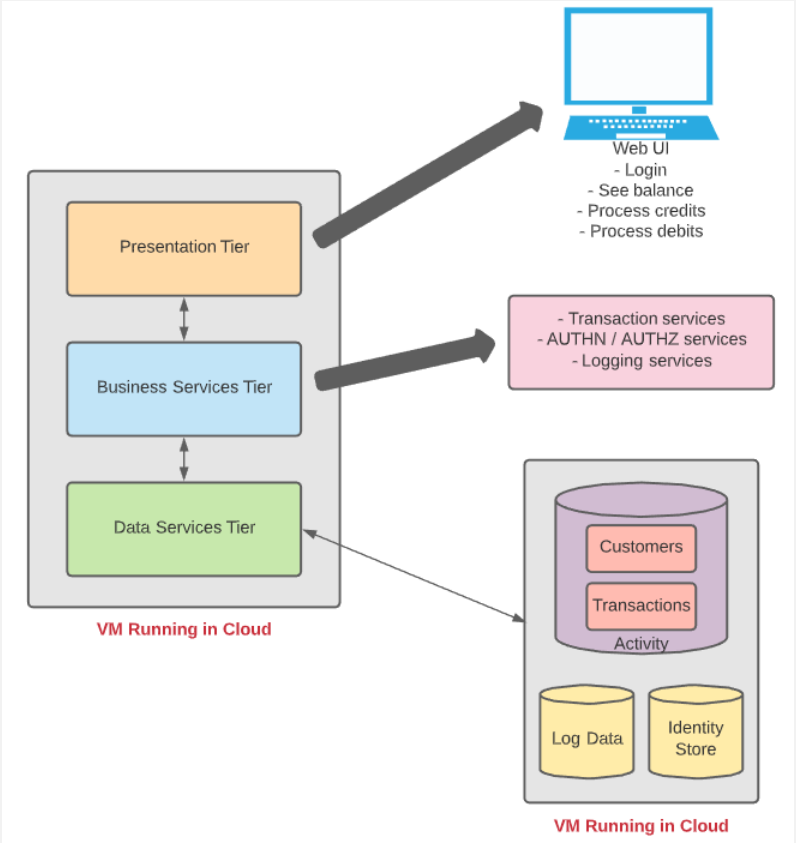


- AKA “lift & shift”
- For all intents and purposes, involves recreating the on-premise infrastructure in the Cloud
- Sometimes used to expedite retirement of a data center



- 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

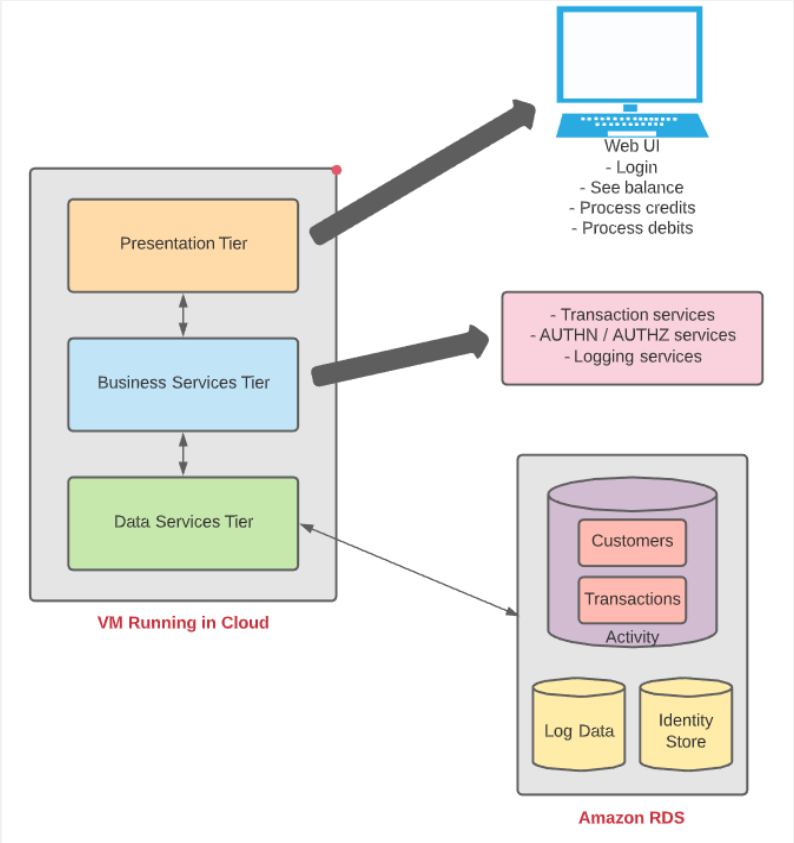

Re-host

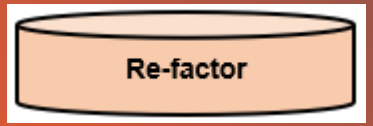

Web UI
- Login
- See balance
- Process credits
- Process debits
Presentation Tier
Business Services Tier
Data Services Tier
- Transaction services
- AUTHN / AUTHZ services
- Logging services
VM Running in Cloud
Customers
Transactions
Activity
Log Data
Identity Store
VM Running in Cloud

- 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

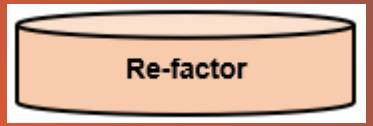
- Those optimizations will involve a move to one or more Cloud native services for a specific, tangible business benefit
- A common example is moving to a managed database (e.g., Relational Database Service, or RDS, in AWS)
- Enables migration speed while providing cost savings or benefit in a targeted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


Re-platform

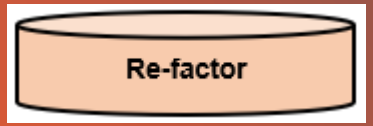

Web UI
- Login
- See balance
- Process credits
- Process debits
Presentation Tier
Business Services Tier
Data Services Tier
VM Running in Cloud
- Transaction services
- AUTHN / AUTHZ services
- Logging services
Customers
Transactions
Activity
Log Data
Identity Store
Amazon RDS



- 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

Reference Architectures

AWS Well-Architected Framework

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

AWS Reference Architectures

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

Azure Reference Architectures – Strangler Application

https://docs.microsoft.com/en-us/azure/architecture/browse/?terms=strangler

Architecting for the Cloud

- The architect must choose between the various types of services provided by the CSP and design optimal integration between those services and components
- It may include a requirement to architect a solution that spans multiple providers or that links on-premise infrastructure & services with cloud-based resources
- Because we lack direct control over Cloud infrastructure and integrations are executed over the Internet, special care needs to be given to architectural concerns like security, resiliency, BC/DR (business continuity/data recovery), and performance
- Additionally, data classification and data sensitivity must be considered and accounted for – what infrastructure (in what areas of the world) is our data flowing through?
- What levels of dedicated connectivity and secrets management are required to ensure that we meet the requirements of our clients and any relevant regulators?

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”

Sample Logical / Physical Architecture Diagrams

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?

Architecture Governance

The Case for Architecture Governance

- As previously mentioned, architecting modern systems requires us to balance competing concerns
- Every approach selected likely corresponds to a set of other options that were dismissed
- Understanding the “why” for a particular decision and vetting that “why” with other peers can help us move forward with confidence
- Additionally, there is value in formalizing a verified solution to a technical problem so that we can reuse that solution when a similar problem arises in the future
- We want to avoid having an enterprise of unicorns

What Is It?

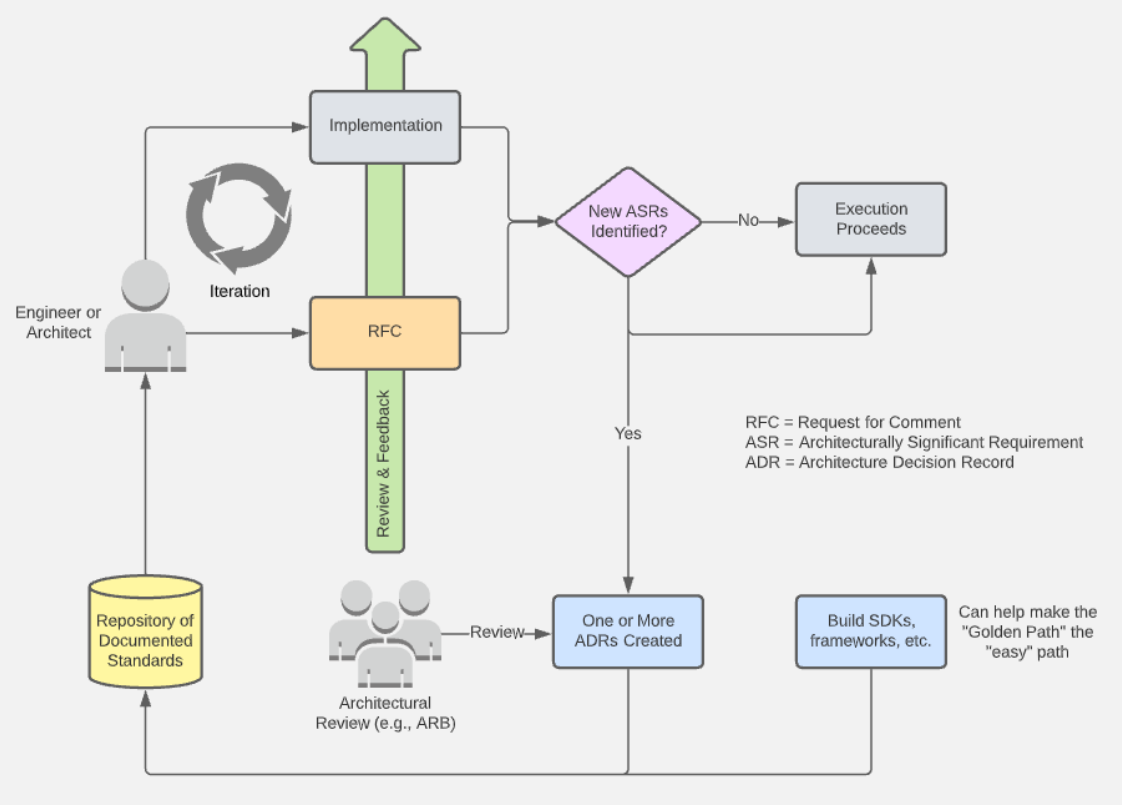
- A formalized process
- Should involve representatives from across the organization (ensure adequate understanding of all perspectives)
- Involves identifying, documenting, and cataloguing architectural standards and best practices
- When a team solves a complex architectural problem, we should codify so others can benefit

What Is It?

- Engineers and architects looking to design software that meets a new business requirement can use this “repository” of standards & best practices for guidance
- Also, formalized reviews of an approach to solving an architecturally significant concern can help to “crowdsource” architectural quality

Think back to the discussion on “fitness functions” – architectural governance can be one of those “fitness functions” that helps guide our change

What Does (Might) it Look Like?



Keys to Remember

- Architecture Governance is not an “ivory tower” from which only select members of the organization decree “from on high”
- Requires a partnership between architecture, engineering, and business – architecture governance is about running architecture “as a business”
- Cross-organizational representation and input helps to ensure less gaps in understanding and a sense of “shared ownership” that can facilitate adherence

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

LAB 09:

Factory Pattern

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab09

Abstract Factory Pattern

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

LAB 10:

Abstract Factory Pattern

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab10

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 11:

Singleton Pattern

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab11

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

LAB 12:

Prototype Pattern

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab12

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 13:

Builder Pattern

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab13

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)

LAB 14:

Adapter Pattern

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab14

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

LAB 15:

Bridge Pattern

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab15

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 16:

Composite Pattern

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab16

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 19:

Flyweight Pattern

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab19

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

LAB 17:

Decorator Pattern

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab17

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 18:

Facade Pattern

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab18

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

LAB 20:

Chain of Responsibility Pattern

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab20

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 21:

Command Pattern

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab21

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

LAB 22:

Observer Pattern

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab22

State Pattern

- Behavioral pattern used to enable changes to a class’s behavior based on its state and state transitions
- Manages state-specific behavior and keeps transitions explicit
- Uses a controller or context to drive behavior based on updates to state

LAB 23:

State Pattern

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab23

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

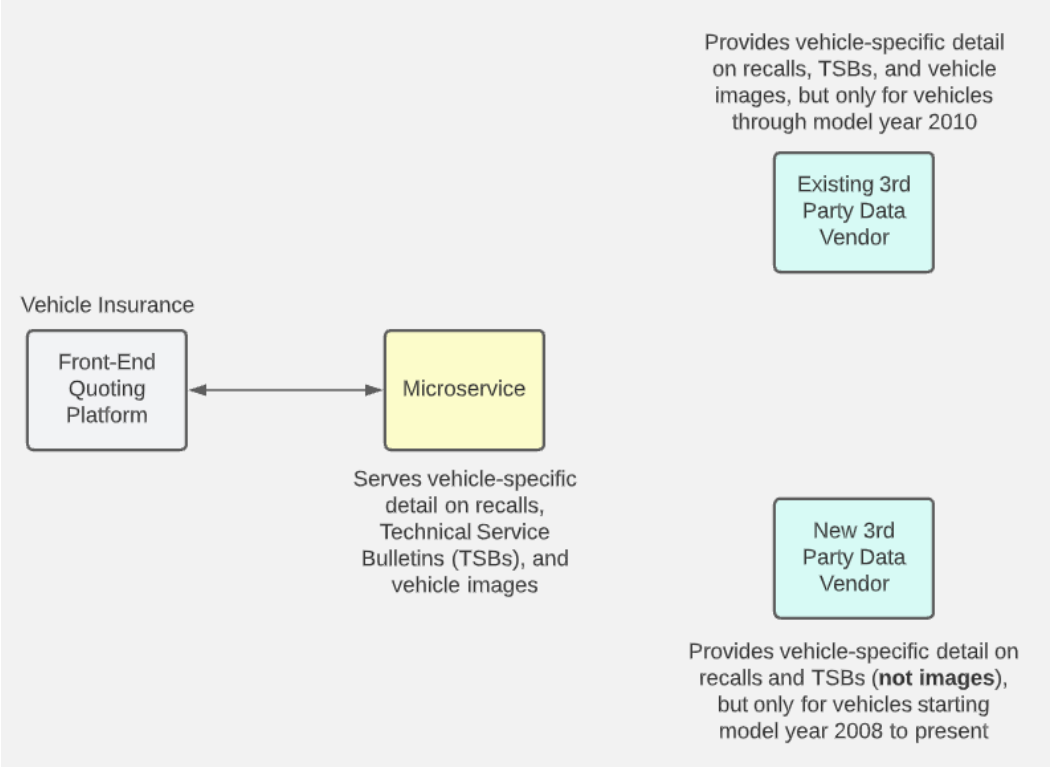
LAB 24:

Strategy Pattern

https://github.com/KernelGamut32/advanced-software-architecture-public/tree/main/labs/lab24

Use Cases

An Exercise in Flexibility



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

SOLID in Practice

https://github.com/KernelGamut32/advanced-software-architecture-public/blob/main/use-cases/Test%20Driven%20Development%20(TDD)%20-%20Class%20Project.pdf

Long Live the Mainframe

https://github.com/KernelGamut32/advanced-software-architecture-public/blob/main/use-cases/Mainframe%20Arch.pdf

It’s the Most Wonderful Time of the Year

https://github.com/KernelGamut32/advanced-software-architecture-public/blob/main/hackathon/Toyshop.pdf

Thank you!

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

DevelopIntelligence
A PLURALSIGHT COMPANY