



**TRAILHEAD**  
TECHNOLOGY PARTNERS

# Avoidifying Over-Complexification

Rooting Out Over-Engineering  
in Your Projects



Jonathan "J." Tower





**TRAILHEAD**  
TECHNOLOGY PARTNERS

credit: scienceworld.scholastic.com

# Identify 10 Types of Over-Engineering

## 10 Rules to Help Avoid It

# Jonathan "J." Tower

Principal Consultant & Partner



**TRAILHEAD**  
TECHNOLOGY PARTNERS

- 🏆 Microsoft MVP in .NET
- ✉️ jtower@trailheadtechnology.com
- 🌐 trailheadtechnology.com/blog
- 🐦 jtowermi
- linkedin jtower

[github.com/trailheadtechnology/over-engineering](https://github.com/trailheadtechnology/over-engineering)

**FREE  
CONSULTATION**



[bit.ly/th-offer](http://bit.ly/th-offer)

# 10 Common Types of Over-Engineering

# Gold-Plating

Over-Engineering Type 1

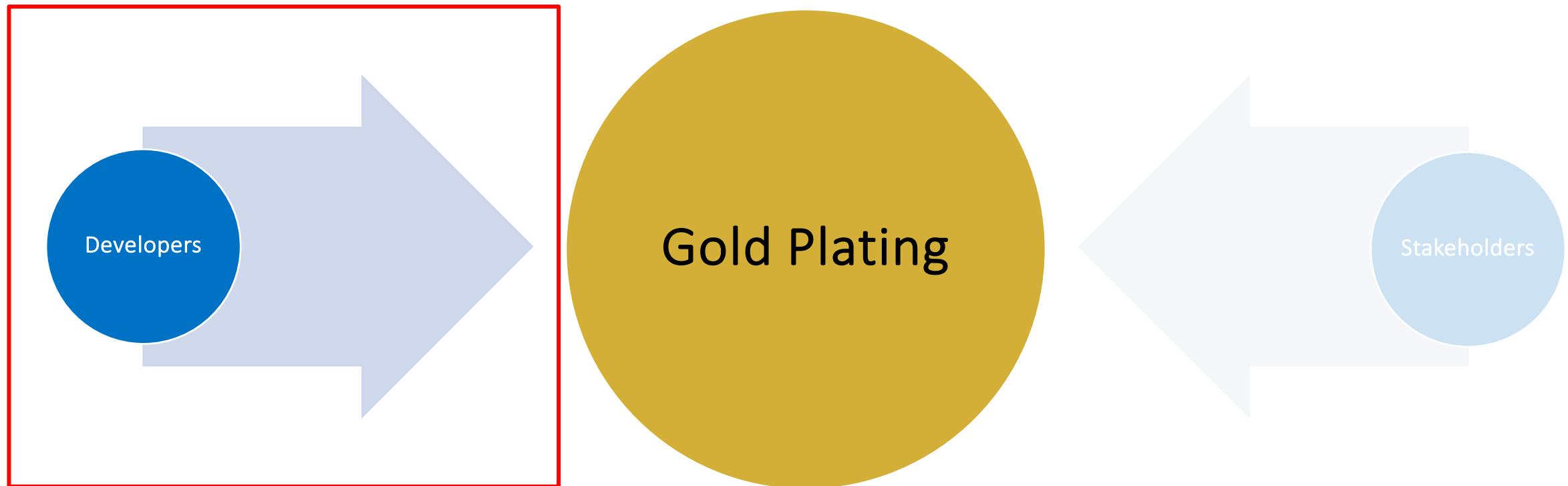


**TRAILHEAD**  
TECHNOLOGY PARTNERS

# Sources of Gold-Plating



# Sources of Gold-Plating



# Reasons Developers Gold-Plate



Perfectionism

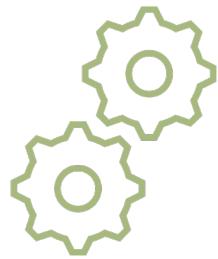


Burned Previously

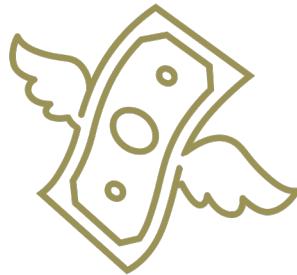


Personal Interest

# Issues With Gold-Plating



Complexity



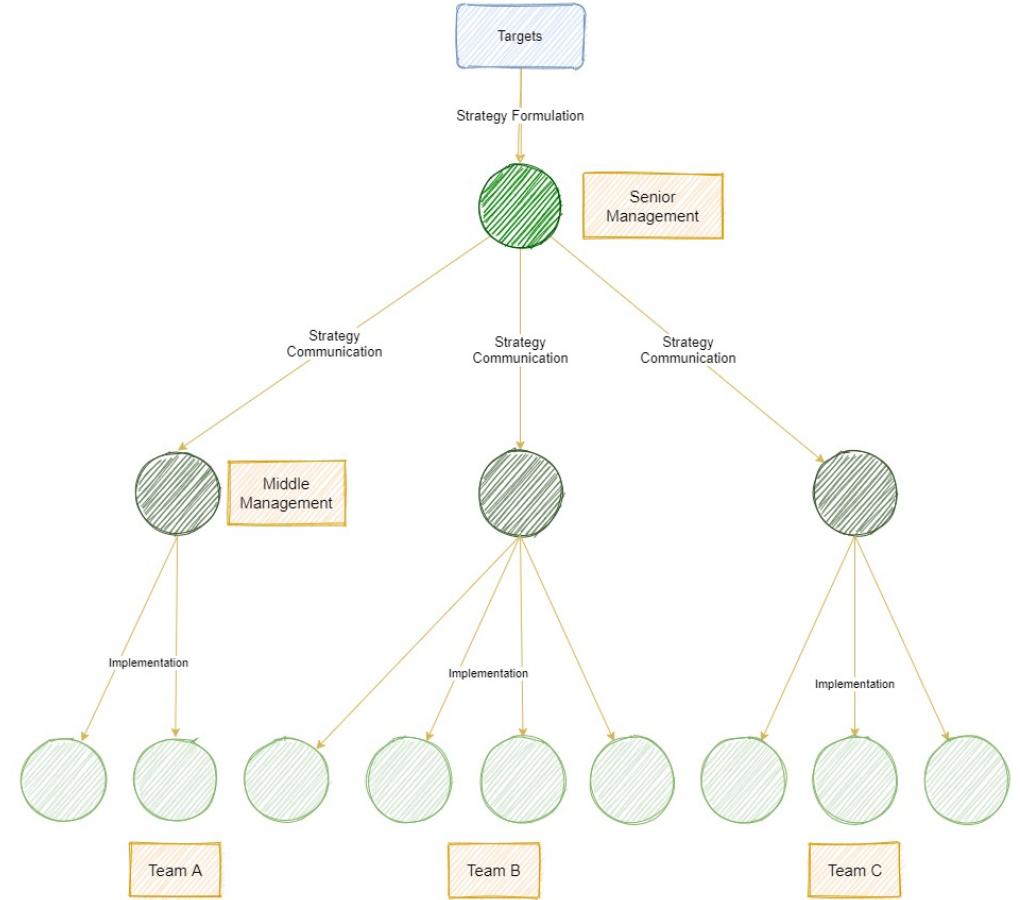
Time & Cost



Loss of Focus

# OO Gymnastics

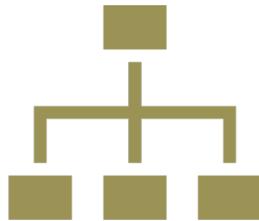
Over-Engineering Type 2



# Types of OO Gymnastics



Unnecessary  
Generics



Complex Inheritance  
Hierarchies



Properties for everything

# Reasons Developers Do OO Gymnastics



Showing Off



Personal Interest

# Over-Abstraction

Over-Engineering Type 3



# Over-Abstraction



# Reasons Developers Over-Abstract



Avoid Vendor Lock-In

# Why You Don't Need Over-Abstraction

- Embrace all 3<sup>rd</sup>-party library has to offer
- Ensure to select libraries that:
  1. Are small
  2. Isolated
  3. Uncomplicated
  4. Replaceable



**Mattias Karlsson (he/him)**  
@devlead

Follow

...

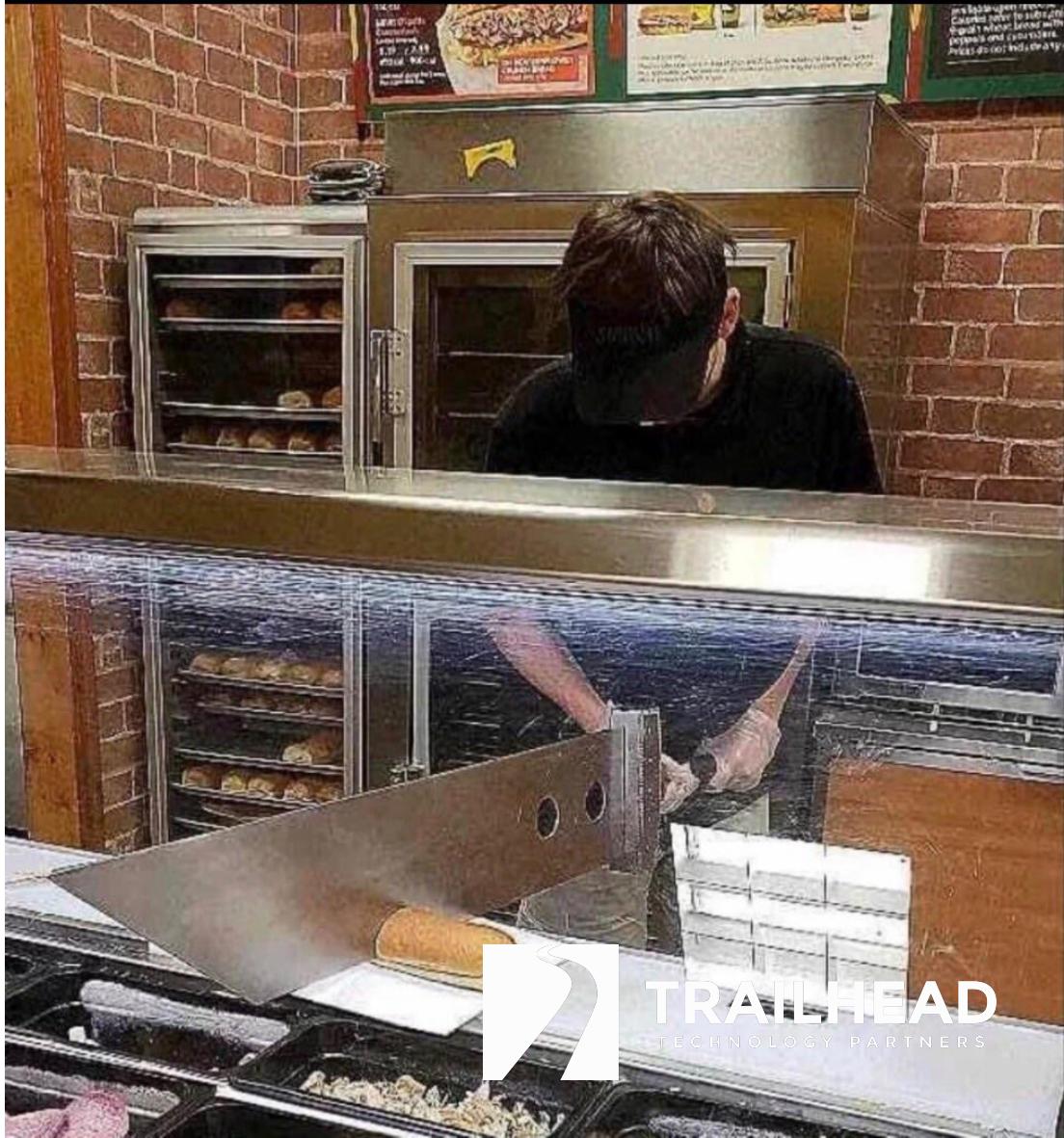
One way to avoid vendor lock-in is to embrace all that the vendor has to offer, but ensure things are small, isolated, uncomplicated, and replaceable.

7:41 AM · Jul 21, 2022

# Over-Built Scalability

Over-Engineering Type 4

When you decide  
to use all your special attacks  
on that level 1 monster



TRAILHEAD  
TECHNOLOGY PARTNERS

# Reasons Developers Over-Build for Scalability



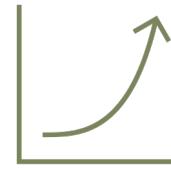
Anticipation of  
Future Needs



Fear of Rework



Influence of  
Trends



Pressure from  
Stakeholders

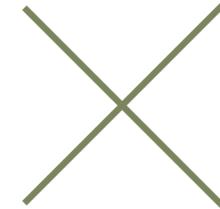
# Issues with Over-Building for Scalability



Increased Time  
& Costs



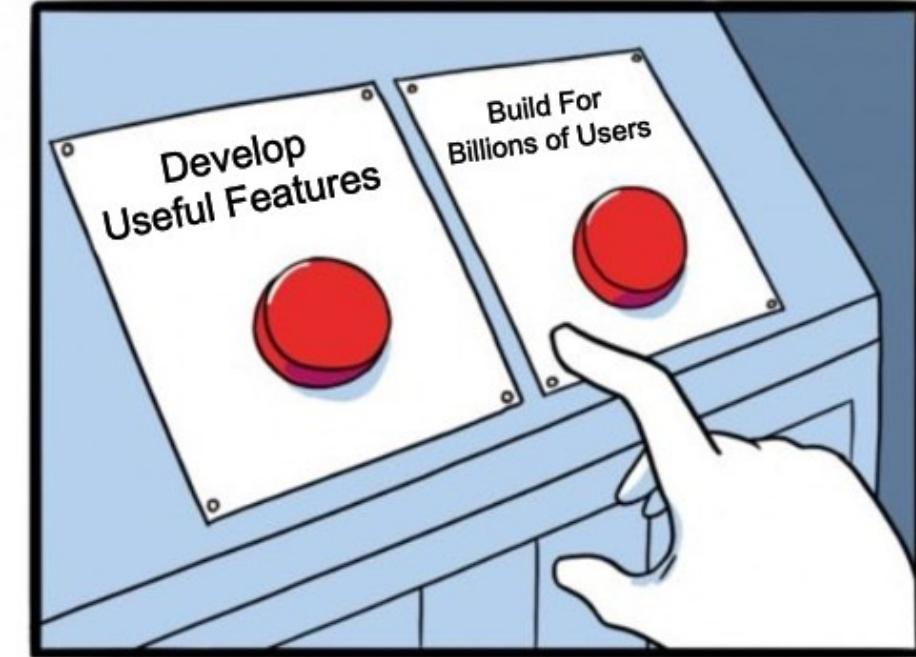
Not Agile



Solving the Wrong  
Problems

# Premature Optimization

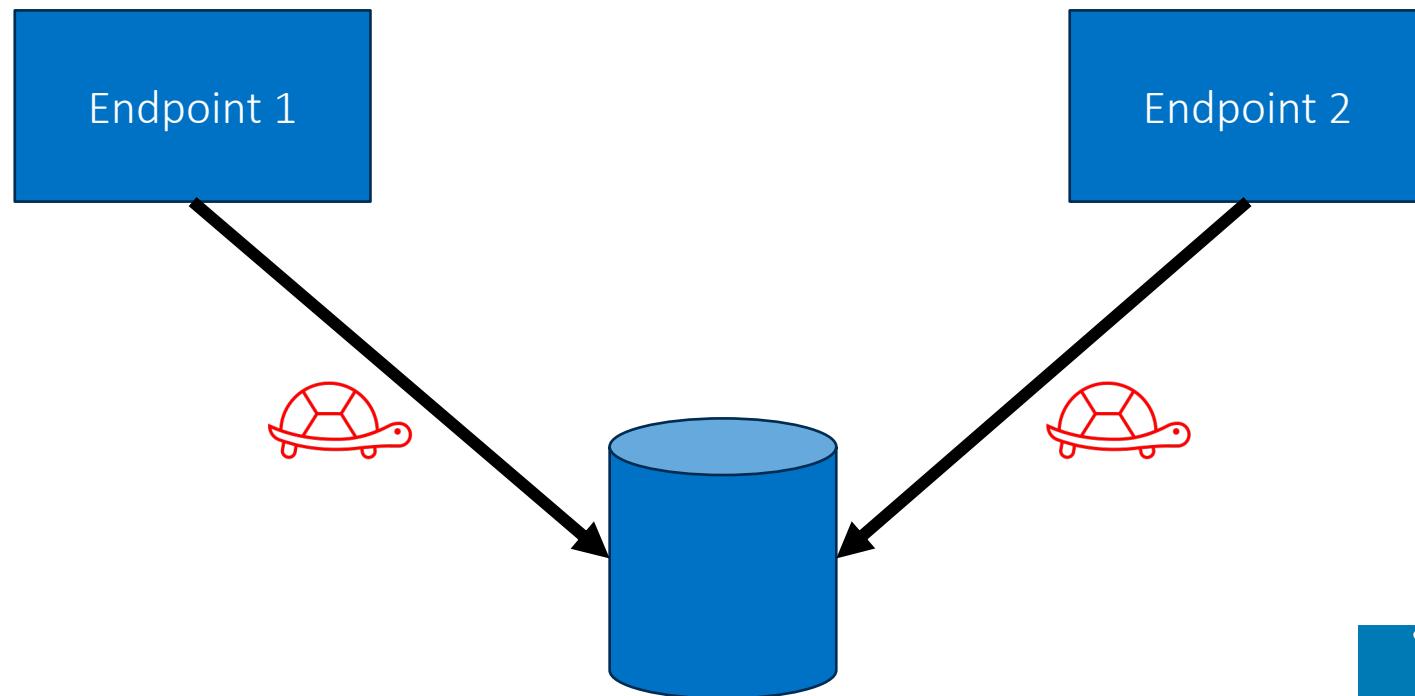
Over-Engineering Type 5



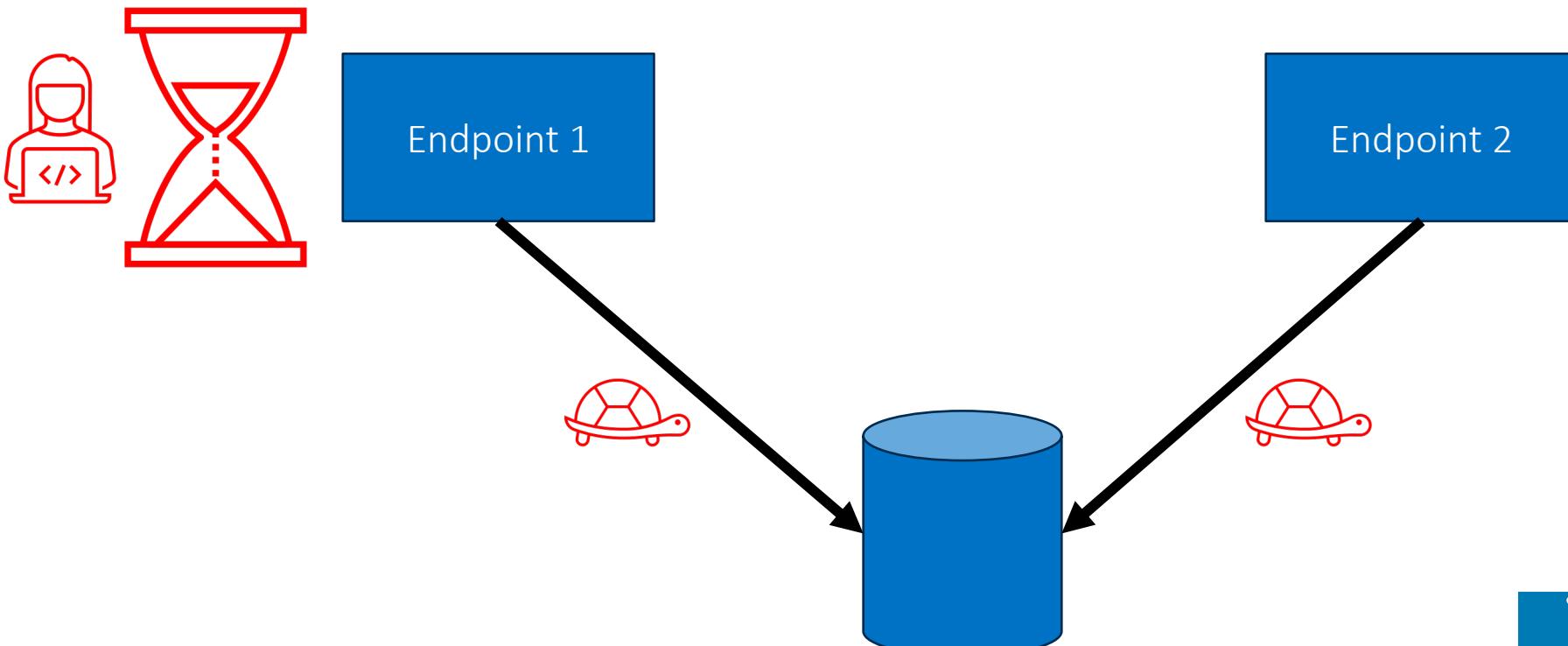
**TRAILHEAD**  
TECHNOLOGY PARTNERS

JAKE-CLARK.TUMBLR

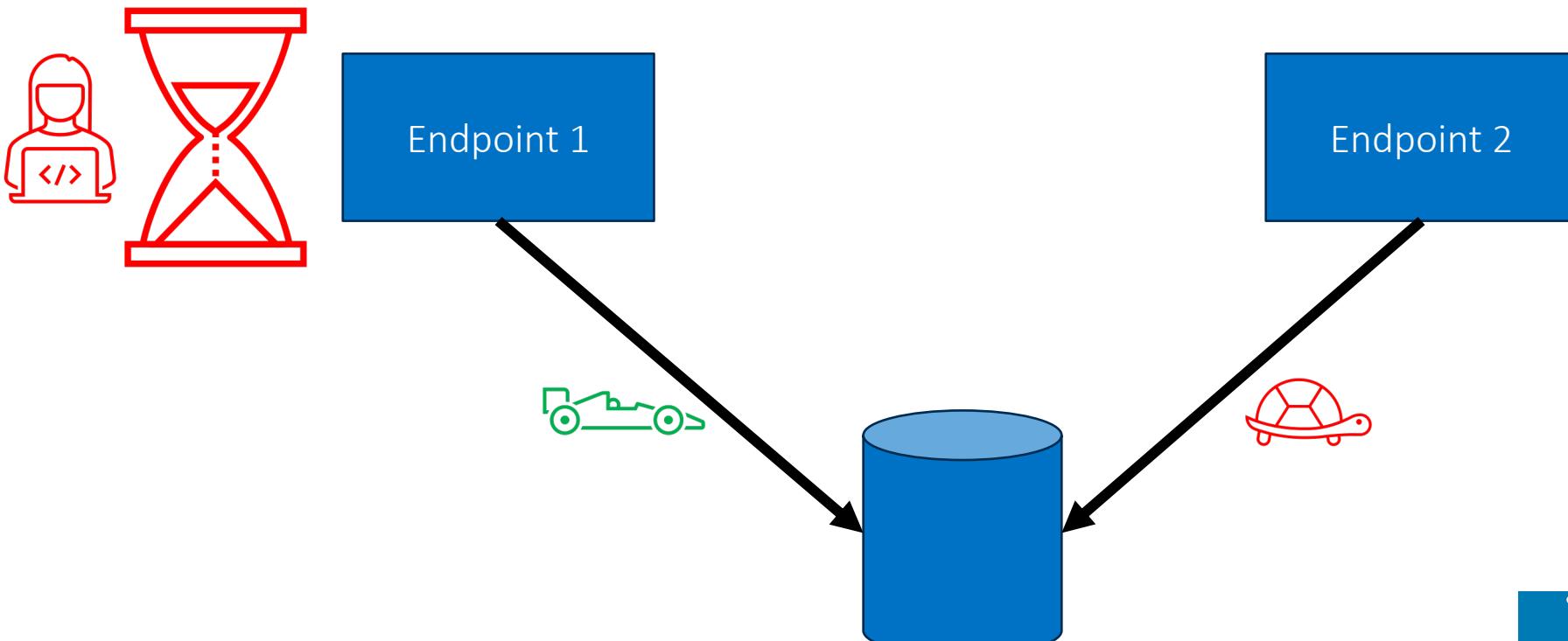
# Premature Optimization



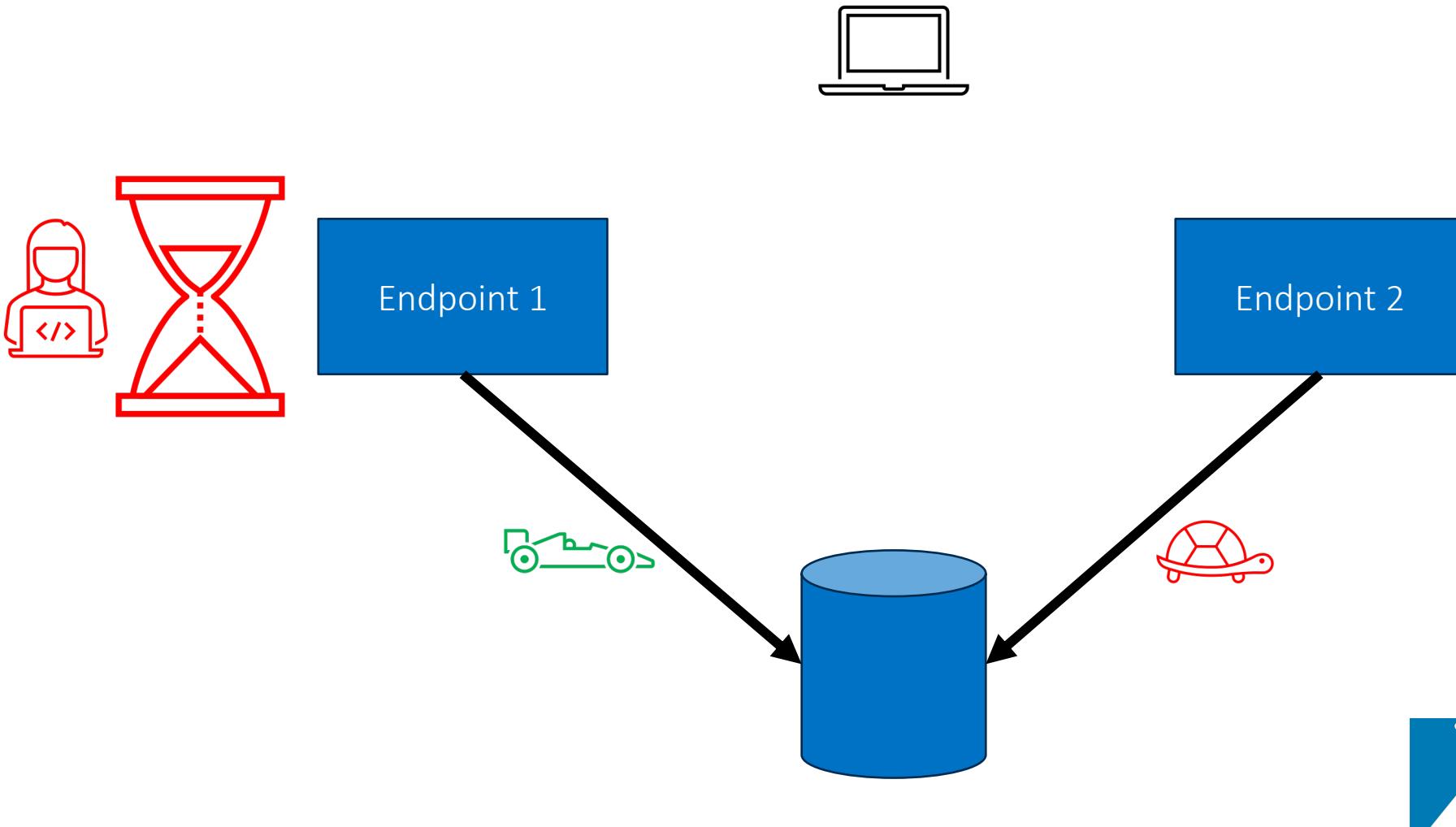
# Premature Optimization



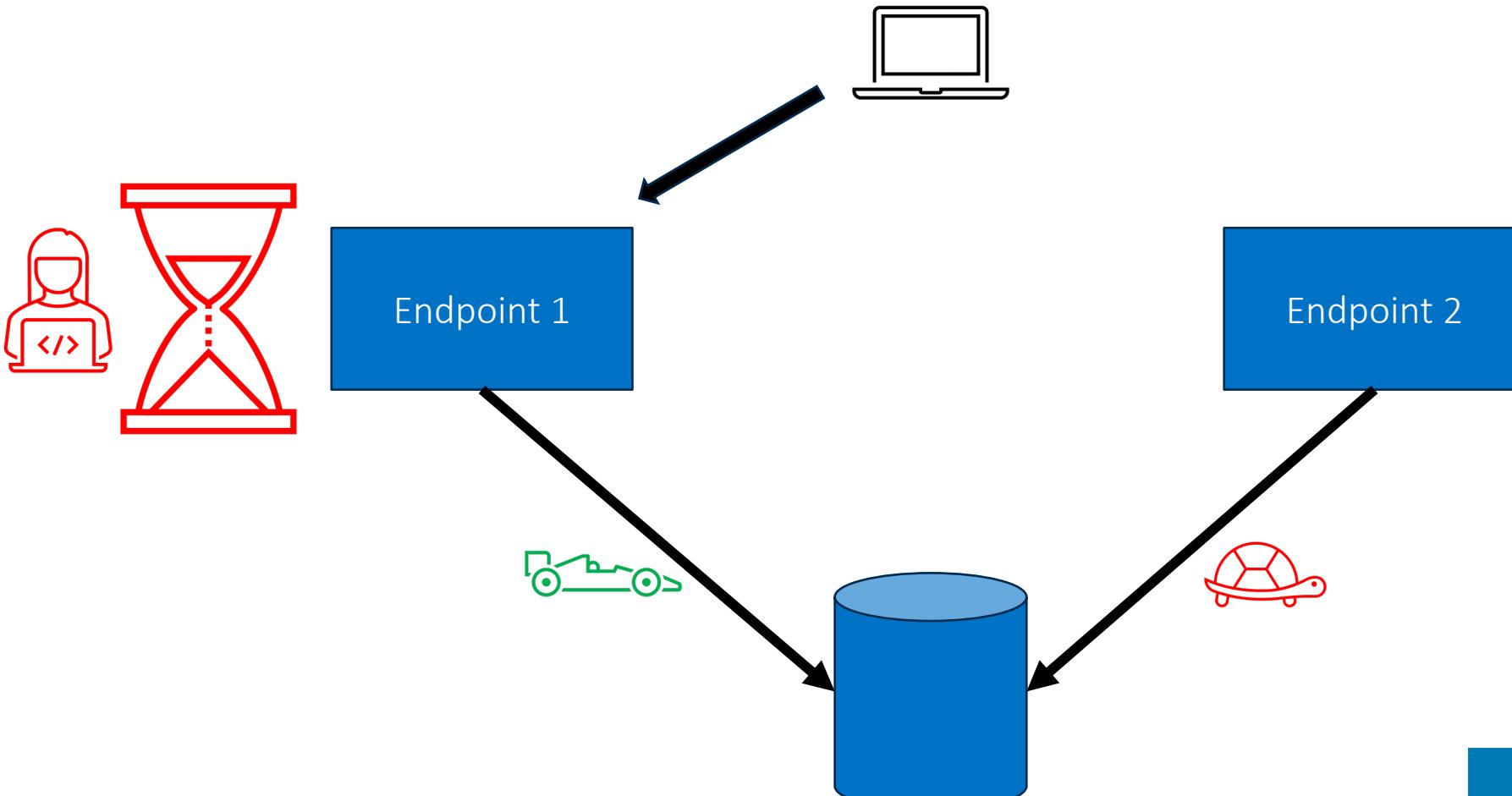
# Premature Optimization



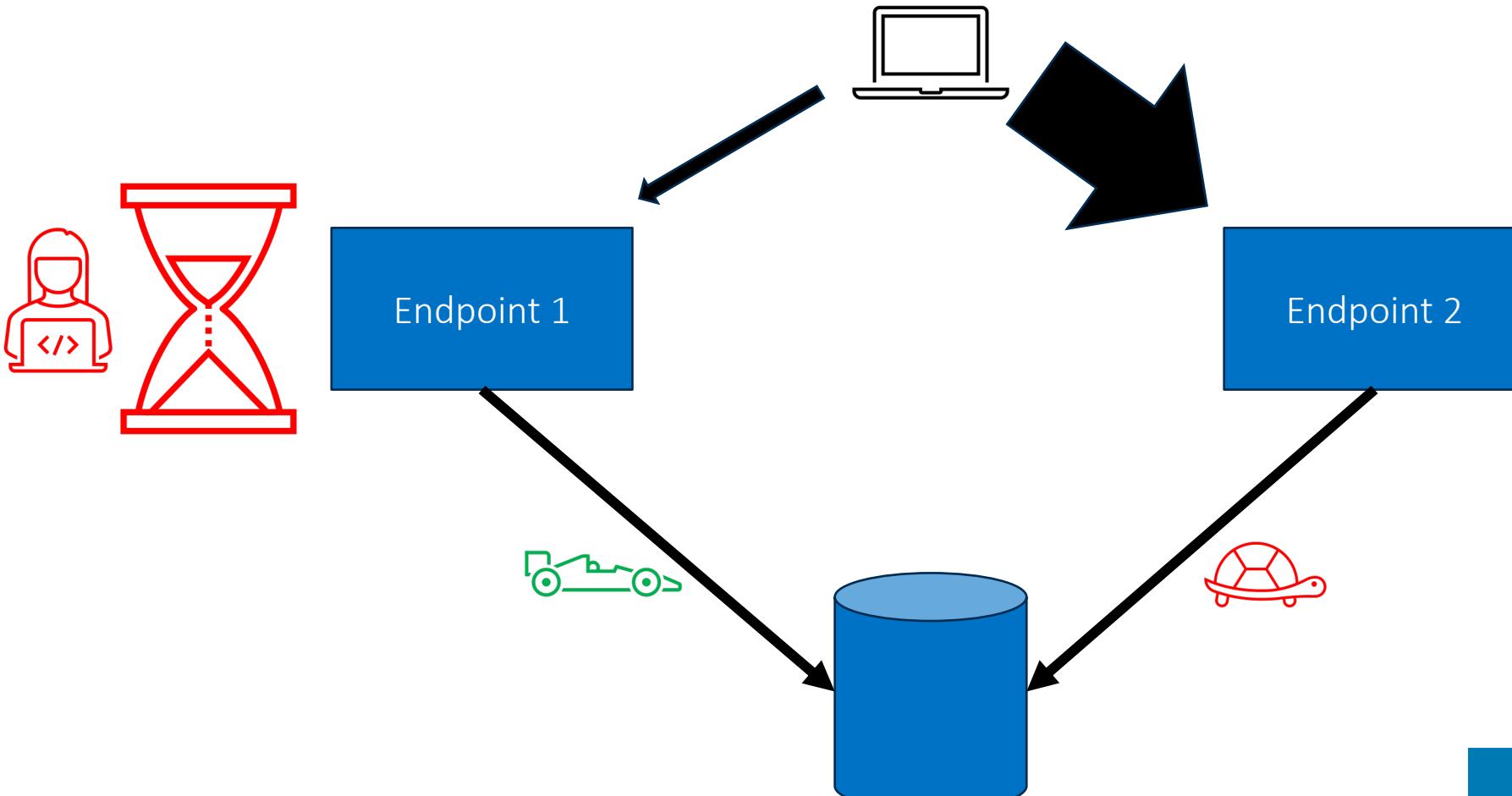
# Premature Optimization



# Premature Optimization



# Premature Optimization



# Reasons Developers Optimize Prematurely



Perfectionism



Assumptions About  
Performance Needs



Fear of Future  
Performance Issues

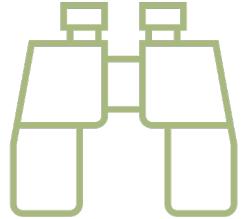


Influence of High-  
Performance Domains



Preemptive Solution  
to Rare Issues

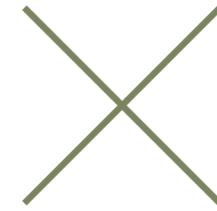
# Issues with Premature Optimization



Increased Costs



Extended  
Development Time



Solving the Wrong  
Problems

# Overuse of Design Patterns

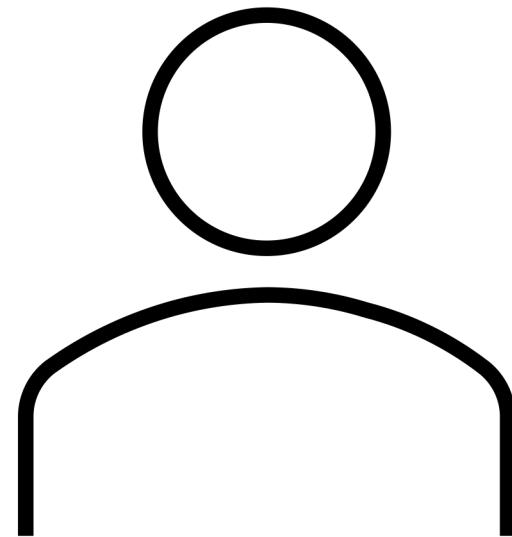
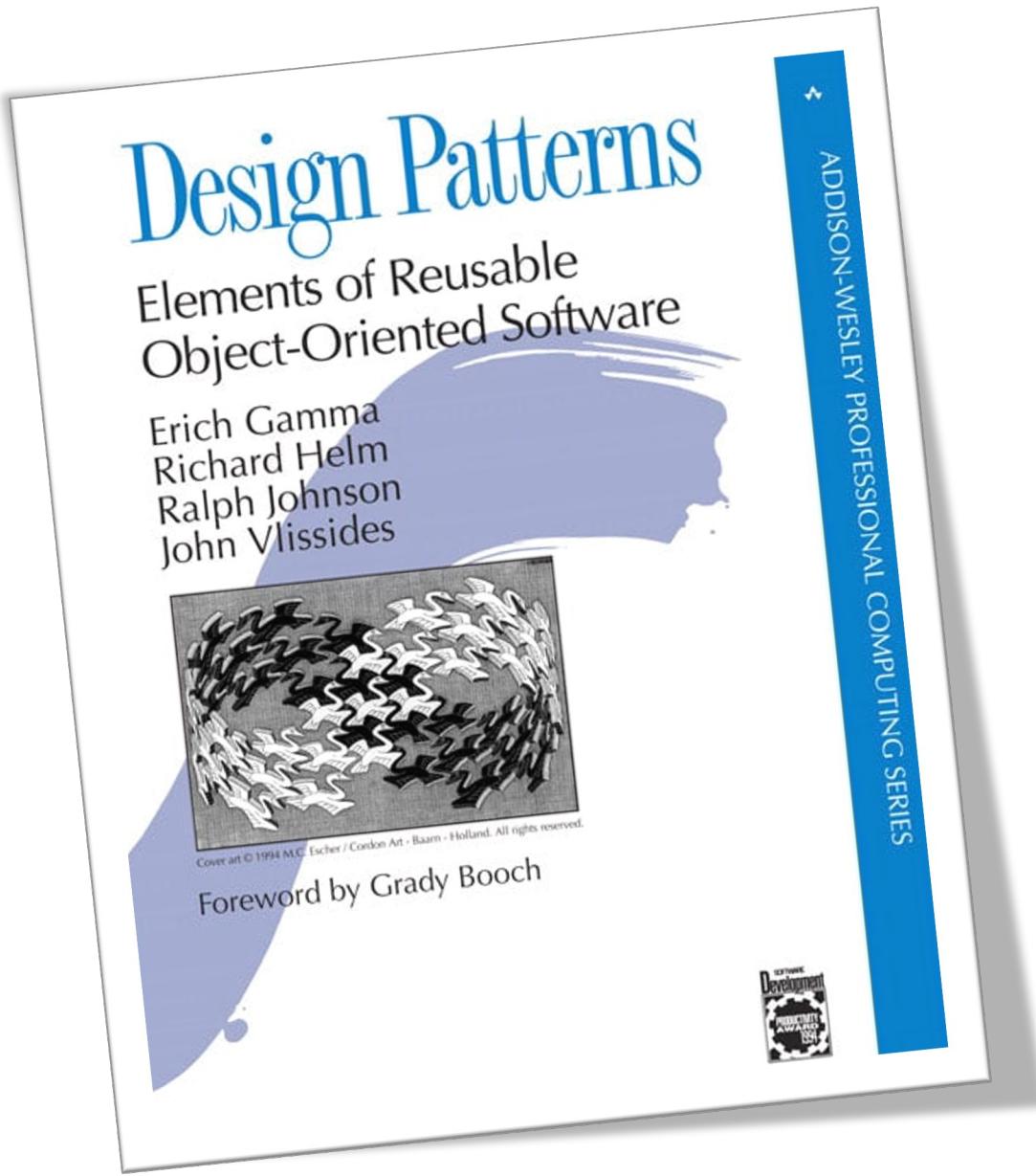
Over-Engineering Type 6

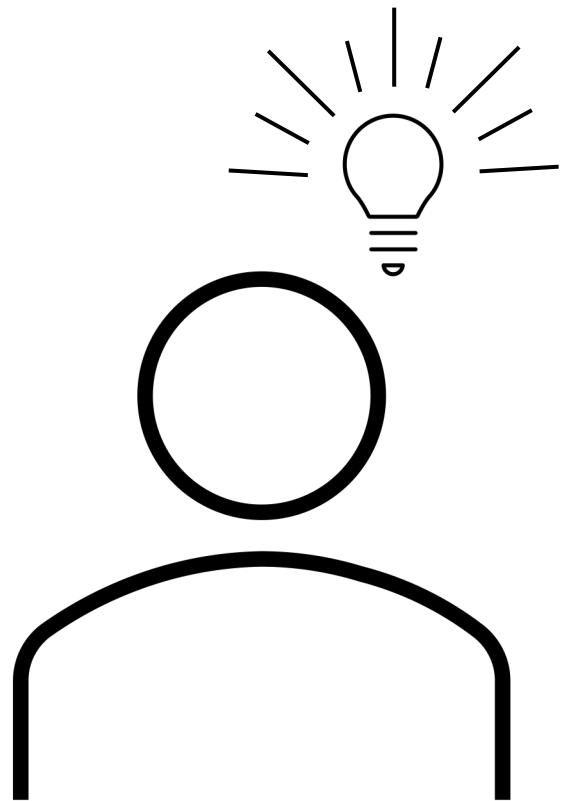


# Design Patterns

Abstract Factory	Facade	Memento
Builder	Flyweight	Null Object
Factory Method	Private Class Data	Observer
Object Pool	Proxy	State
Prototype	Chain of responsibility	Strategy
Singleton	Command	Template method
Adapter	Interpreter	Visitor
Bridge	Iterator	
Composite	Mediator	
Decorator		







**“If the only  
tool you have is  
a hammer, it is  
tempting to  
treat **everything**  
as if it were a  
nail.”**

- Abraham Maslow



# Reasons Developers Overuse Design Patterns



Educational  
Enthusiasm



Misunderstanding of  
Design Patterns



Desire to Follow Best  
Practices



Peer Pressure and  
Industry Trends



Fear of Code Smells

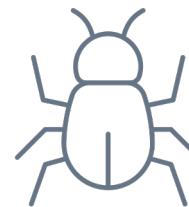
# Issues with Overuse of Design Patterns



Increased Costs



Extended  
Development  
Time



Higher Risk Of  
Bugs



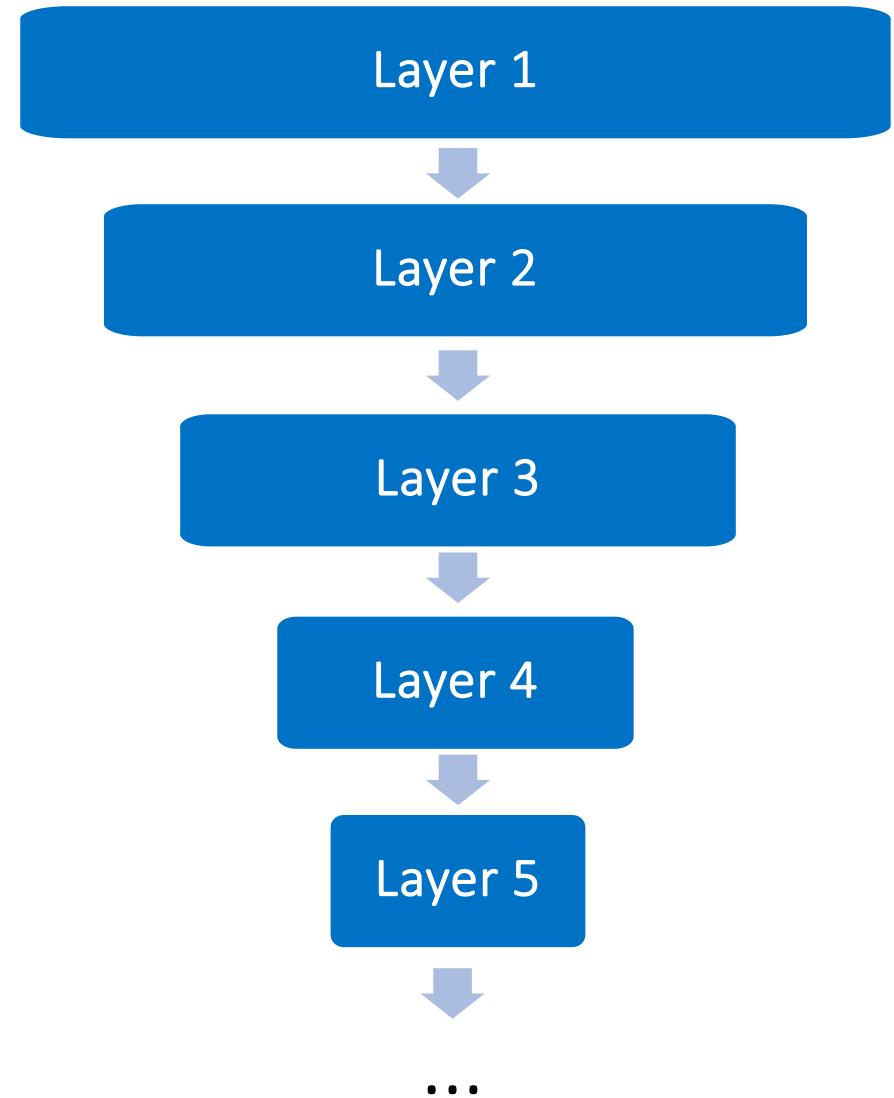
Slowing New  
Developers

# Lasagna Architecture

Over-Engineering Type 7



# Too Many Layers



# Vertical Slice Architecture

Vertical slice architecture is a **software architecture** pattern that **organizes** code **by features** or use cases **instead of technical** concerns.

*No more layers for their own sake!*

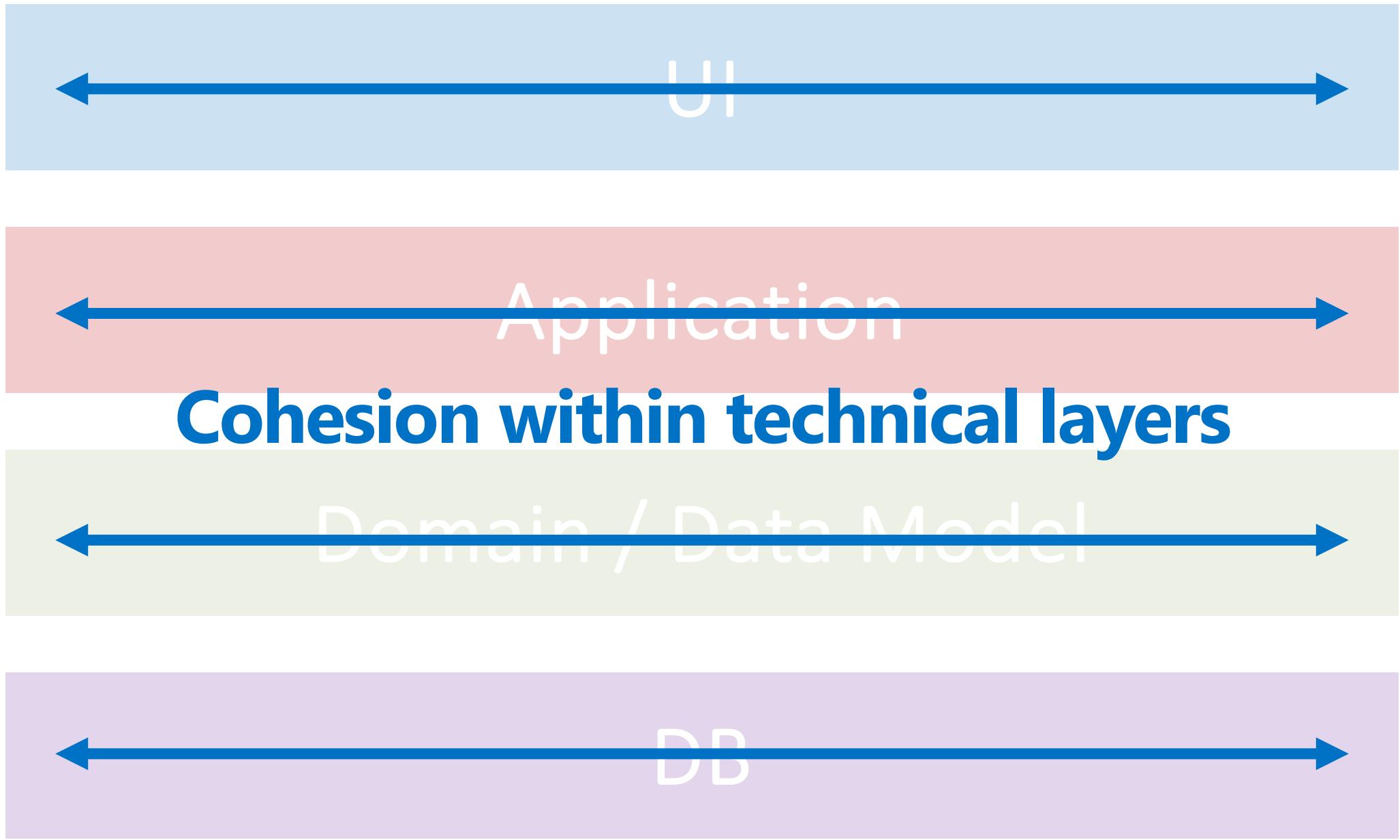


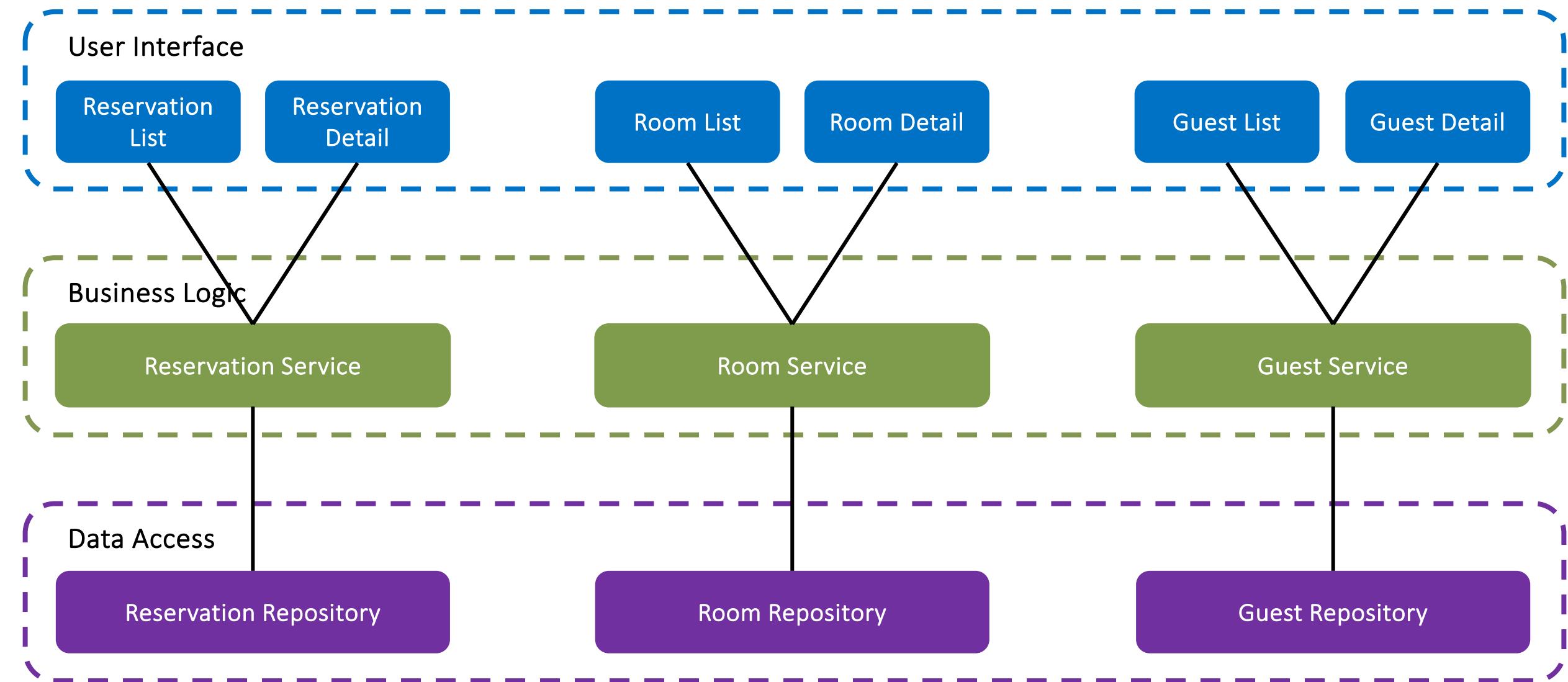
UI

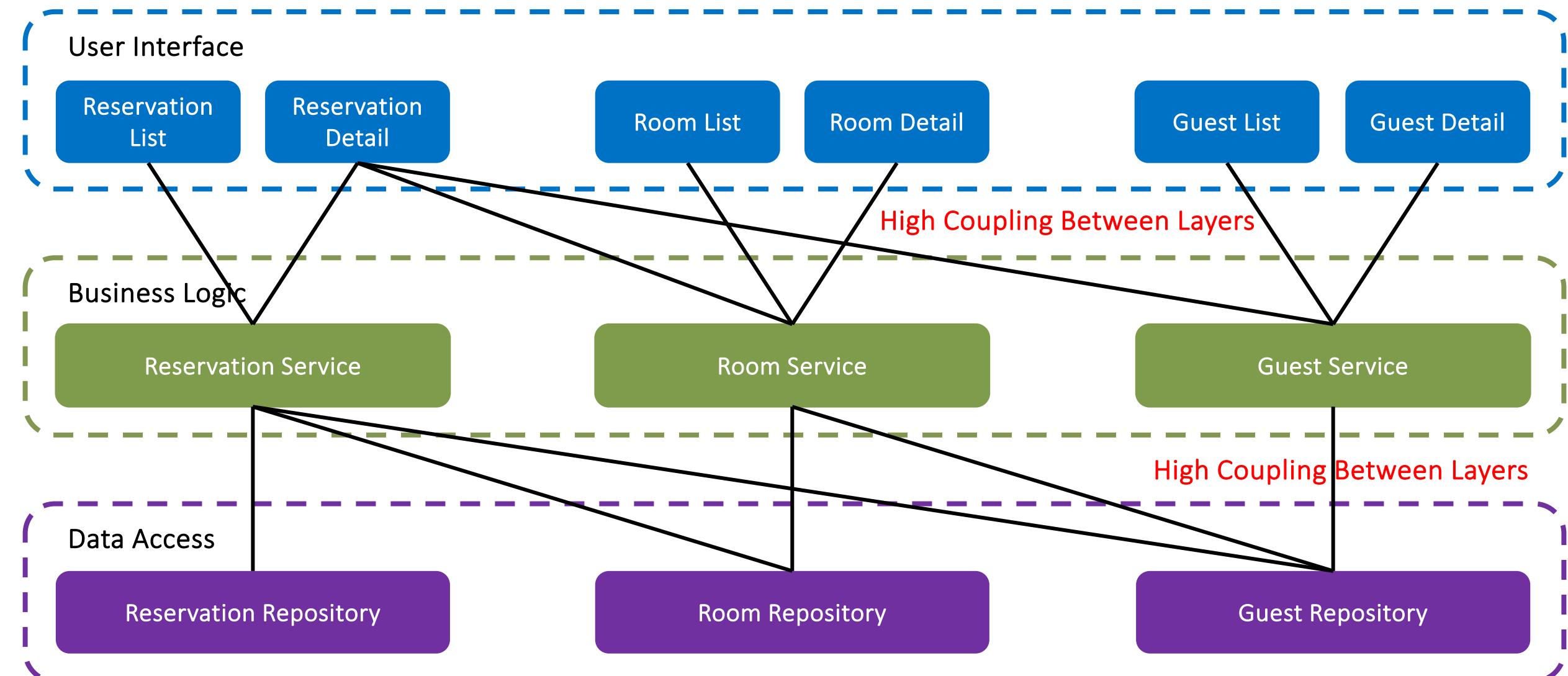
Application

Domain / Data Model

DB







# Coupling Between Layers and Features

User Interface

Reservation List

Reservation Detail

Business Logic

Reservation Service

Room List

Room Detail

Guest List

Guest Detail

Data Access

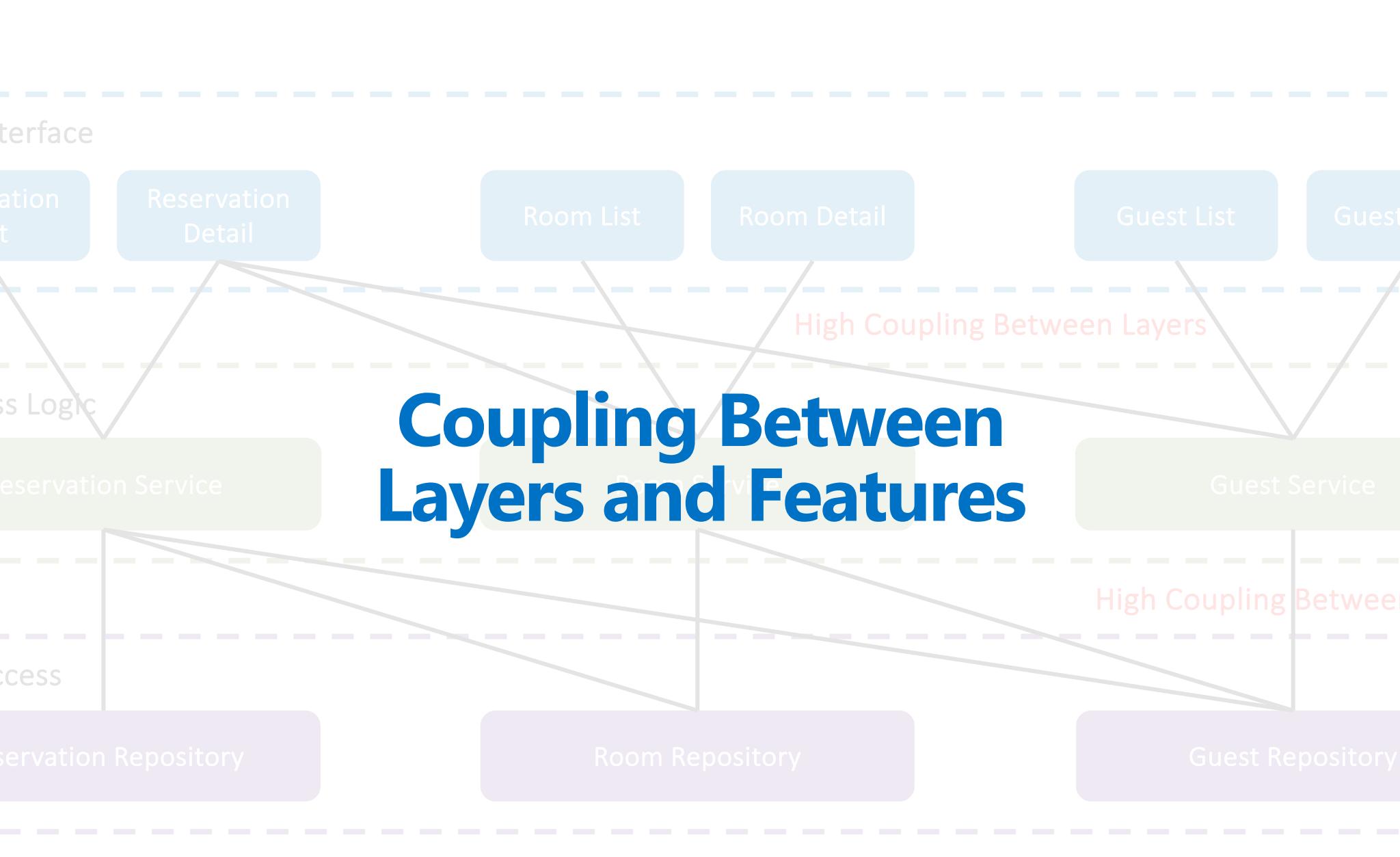
Reservation Repository

Room Repository

Guest Repository

High Coupling Between Layers

High Coupling Between Layers



UI

Application

Domain / Data Model

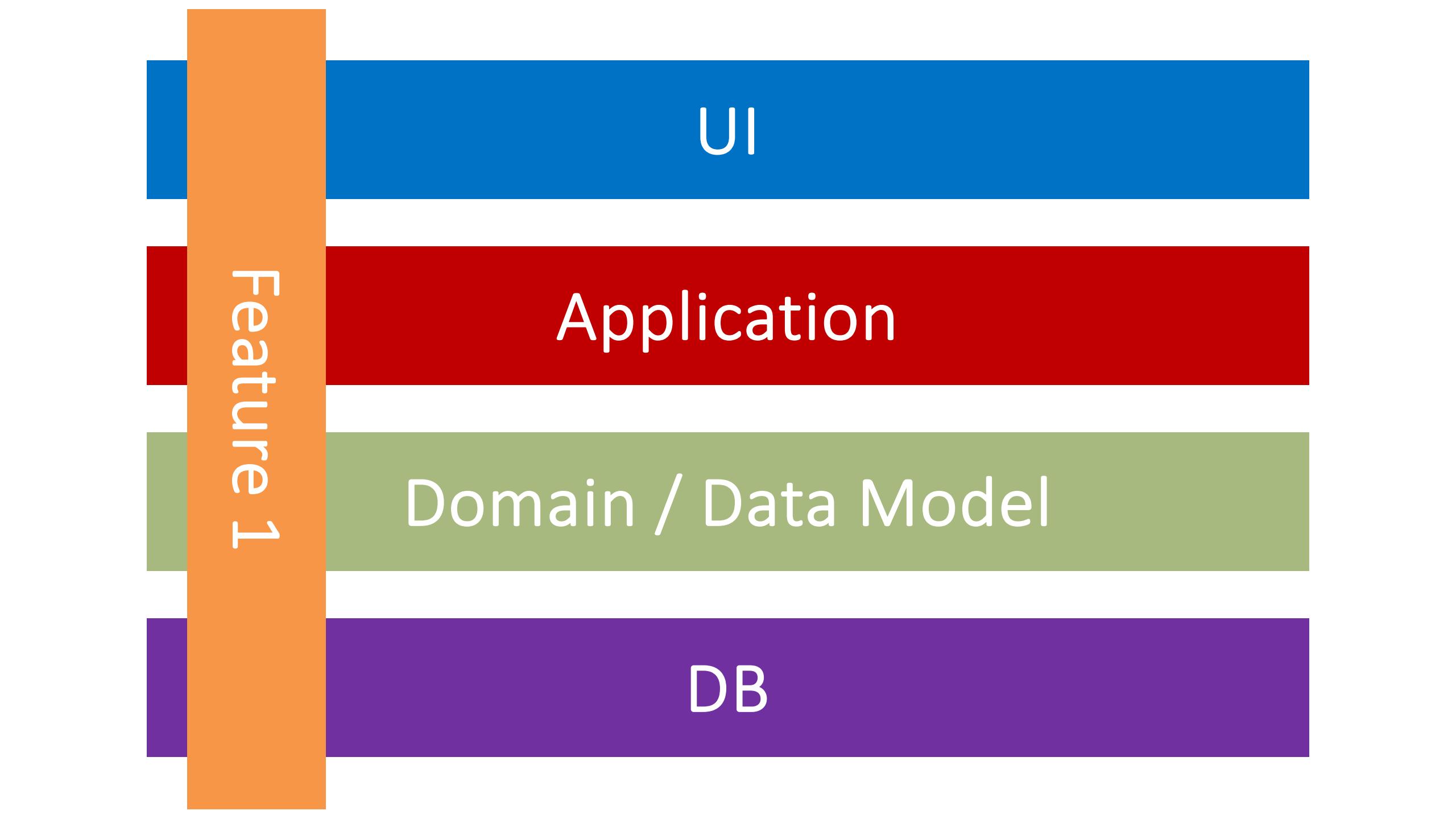
DB

UI

Application

Domain / Data Model

DB



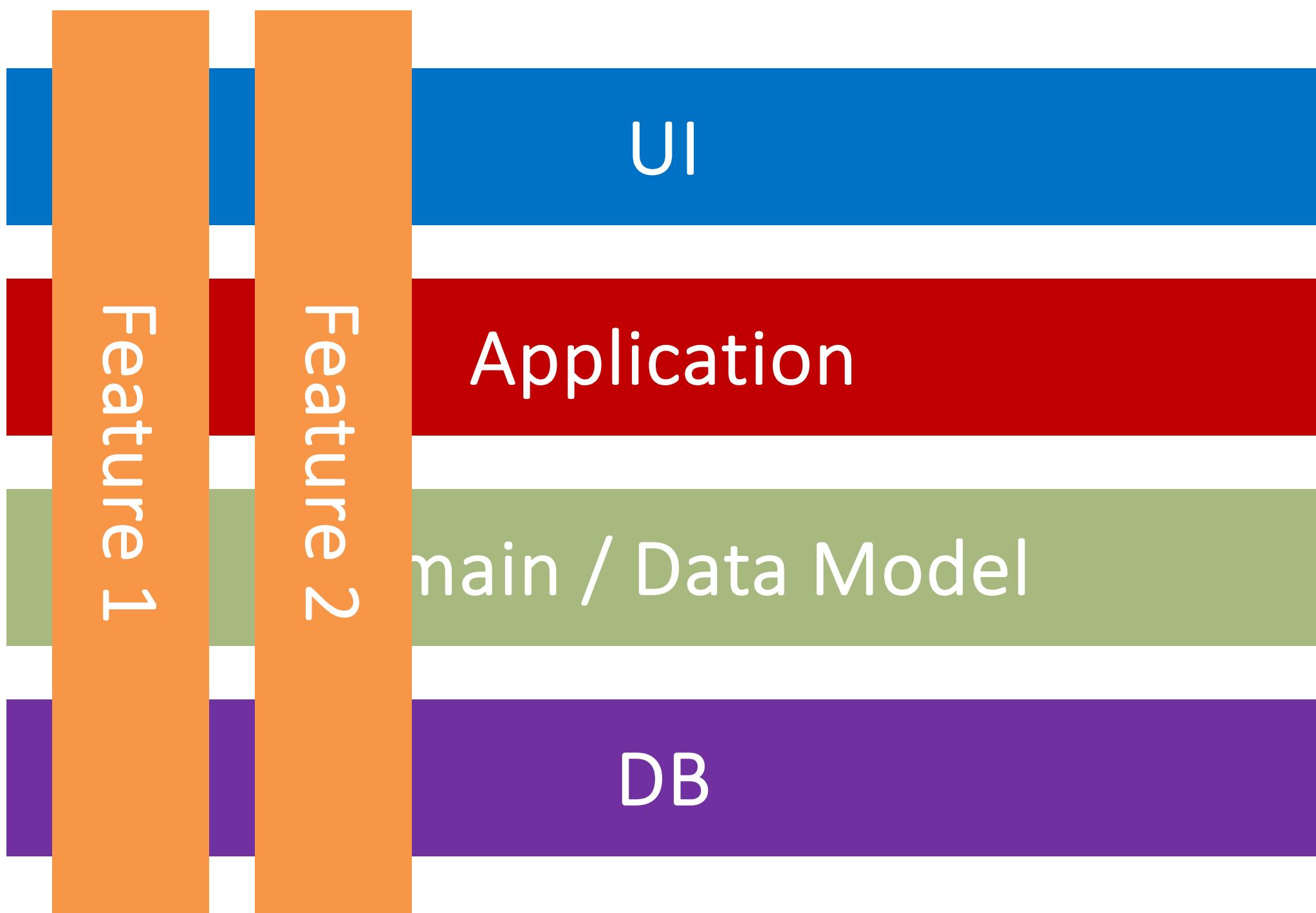
UI

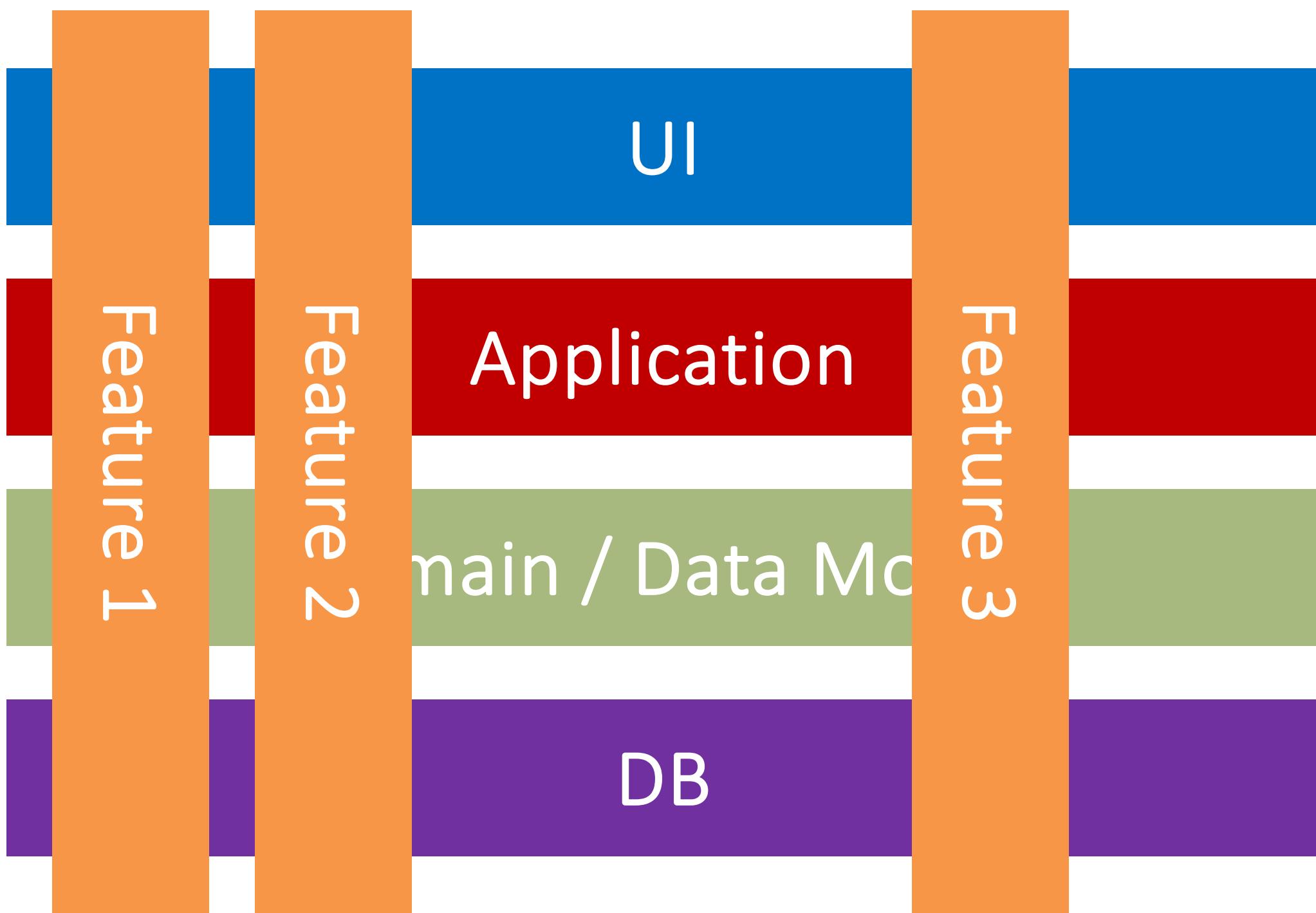
Application

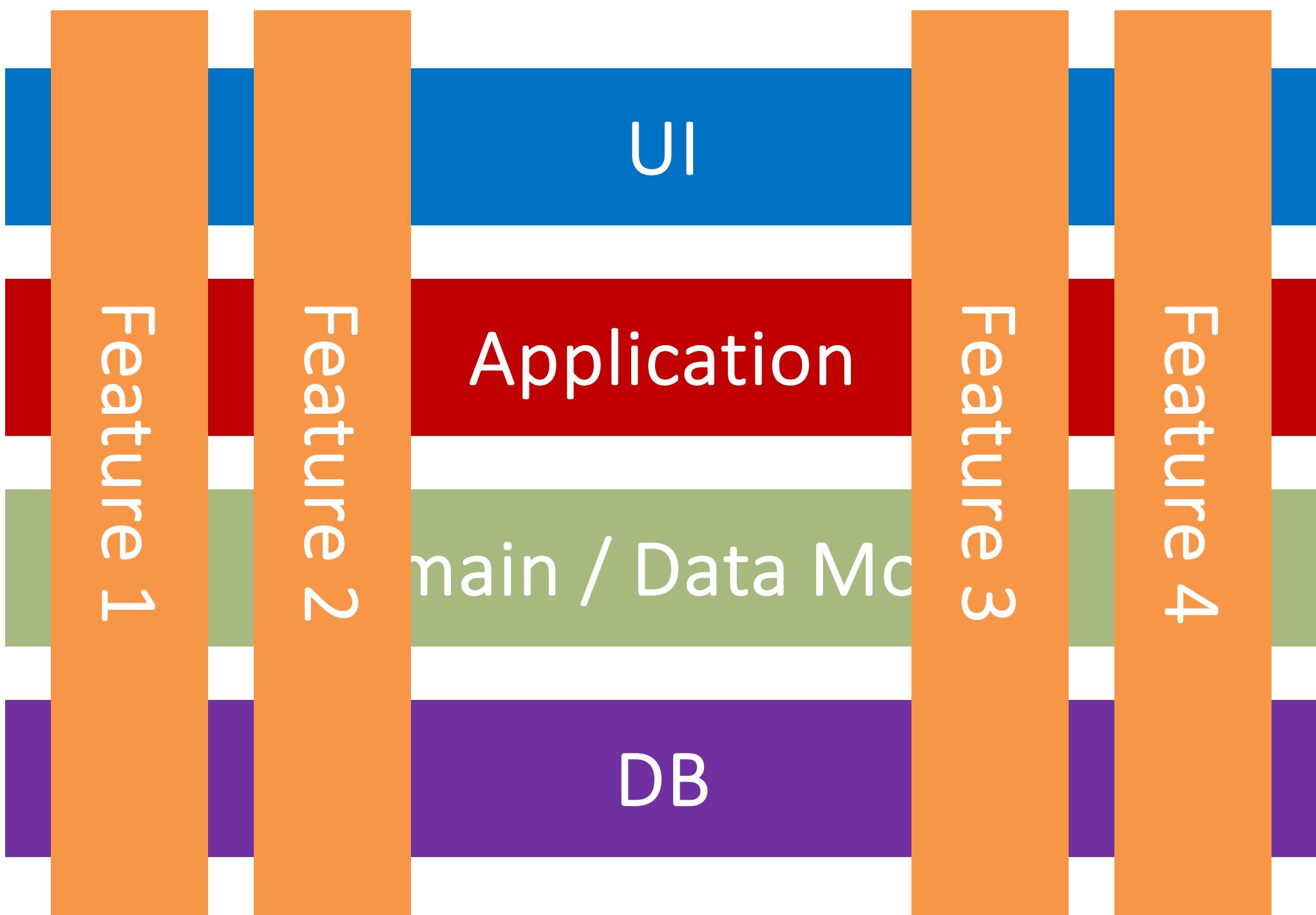
Domain / Data Model

DB

Feature 1







UI

Application  
**Cohesion**  
is within a  
**feature**

DB

Feature 1

Feature 2

Feature 3

Feature 4

Application  
**Cohesion**  
is within a  
**feature**  
**vertical slice**

UI

DB

Feature 1

Feature 2

Feature 3

Feature 4

# Shiny Object Syndrome

Over-Engineering Type 8



# Which Stack Is More Fun?

UI

Middle-Tier

Backend

Stuff You Already Know

NEW!

UI

Middle-Tier

NEW!

Backend

All New Stuff

# Reasons Developers Go For Shiny Objects



Curiosity and  
Enthusiasm



Perceived  
Prestige



Peer Pressure



Fear of  
Missing Out



Marketing and  
Hype

# Issues with Shiny Object Syndrome



Wrong Tool



Rewrite



Hard to Maintain

# Rolling Your Own

Over-Engineering Type 9



**TRAILHEAD**  
TECHNOLOGY PARTNERS

# Reasons Developers Roll Their Own



Perceived Unique  
Requirements



Desire for Control



Lack of Awareness



Integration Issues



Performance  
Optimization

# Issues with Rolling Your Own



Increased Costs



Extended  
Development  
Time



Lack of Expertise



Reduced Testing  
Exposure

# Misapplied Libraries

Over-Engineering Type 10



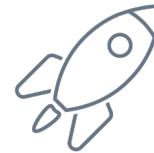
# Reasons Developers Misapply Libraries



Lack of  
Understanding



Overconfidence



Overcompensation



Fear of Simplicity



Educational Gap

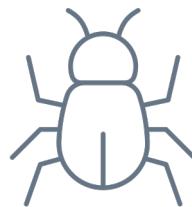
# Issues with Misapplied Libraries



Increased Costs



Extended  
Development  
Time



Higher Risk Of  
Bugs



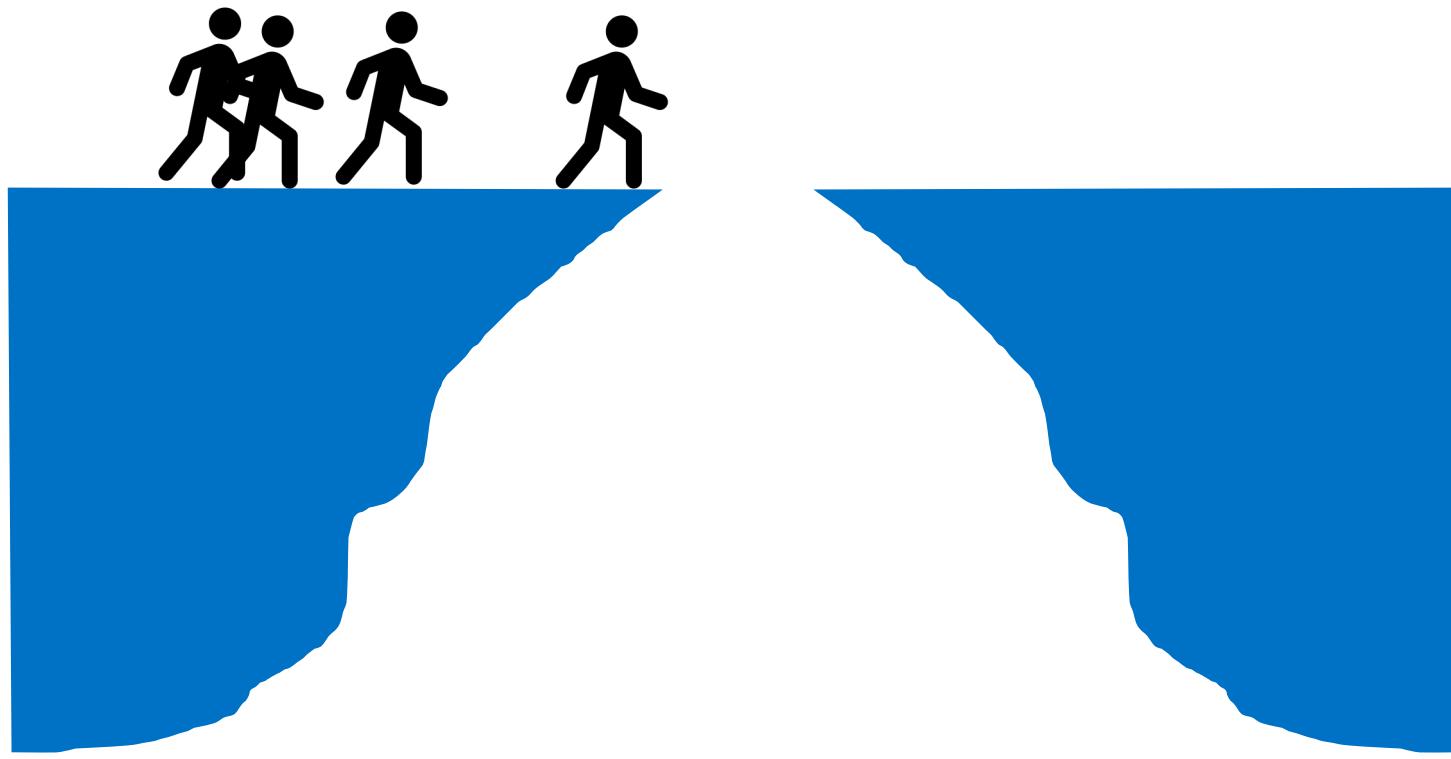
Slowing New  
Developers

# 10 Rules To Avoid Over-Engineering

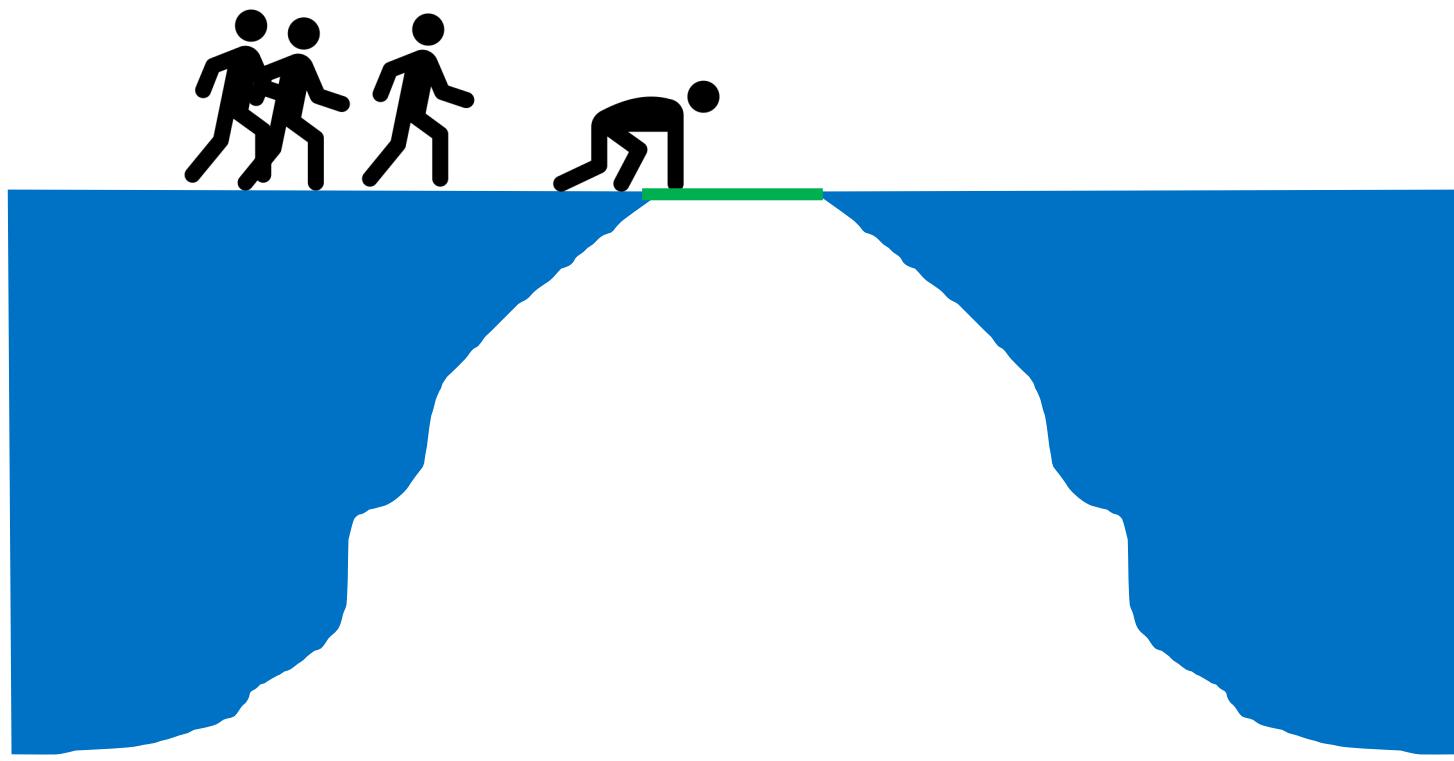
# Document Engineering Decisions

Rule 1

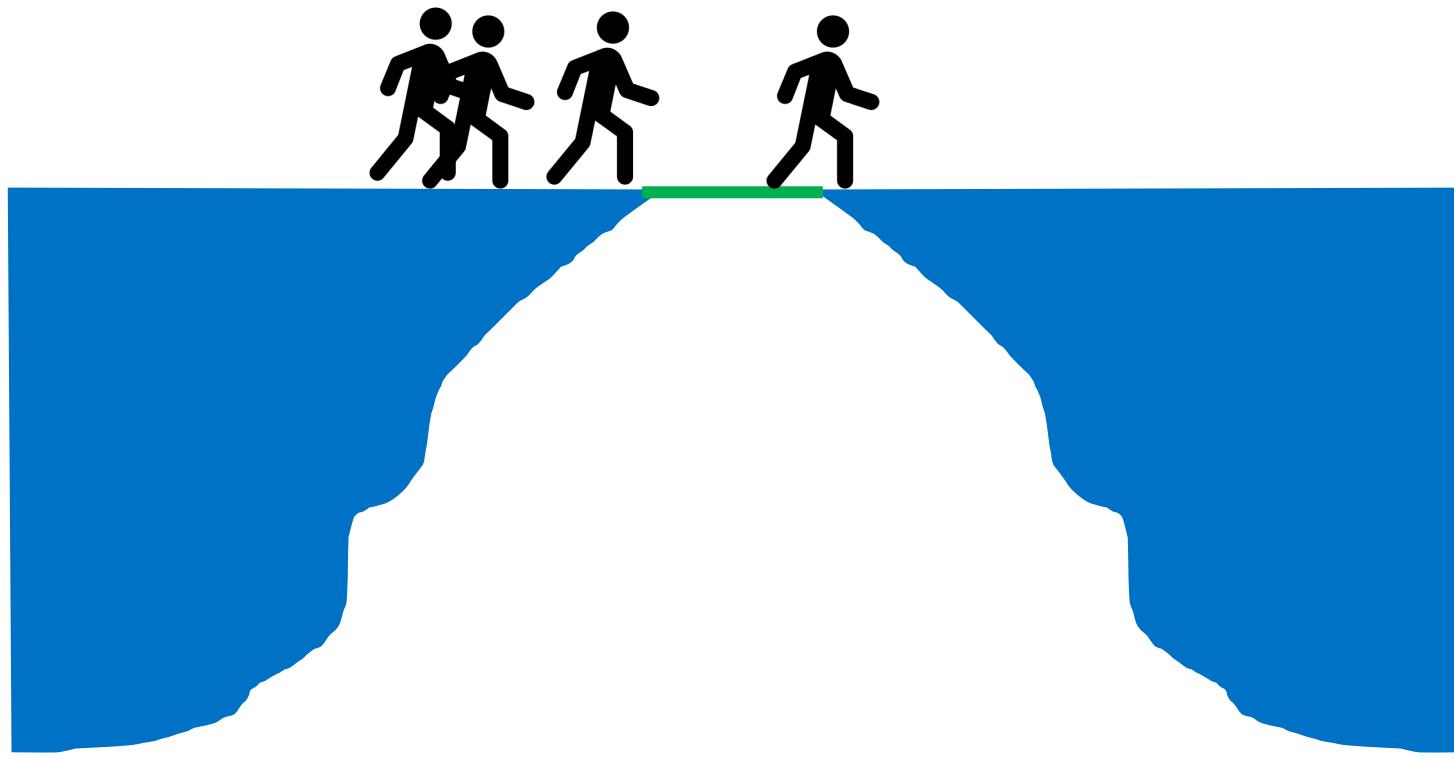
# Avoid Mistakes In The Future



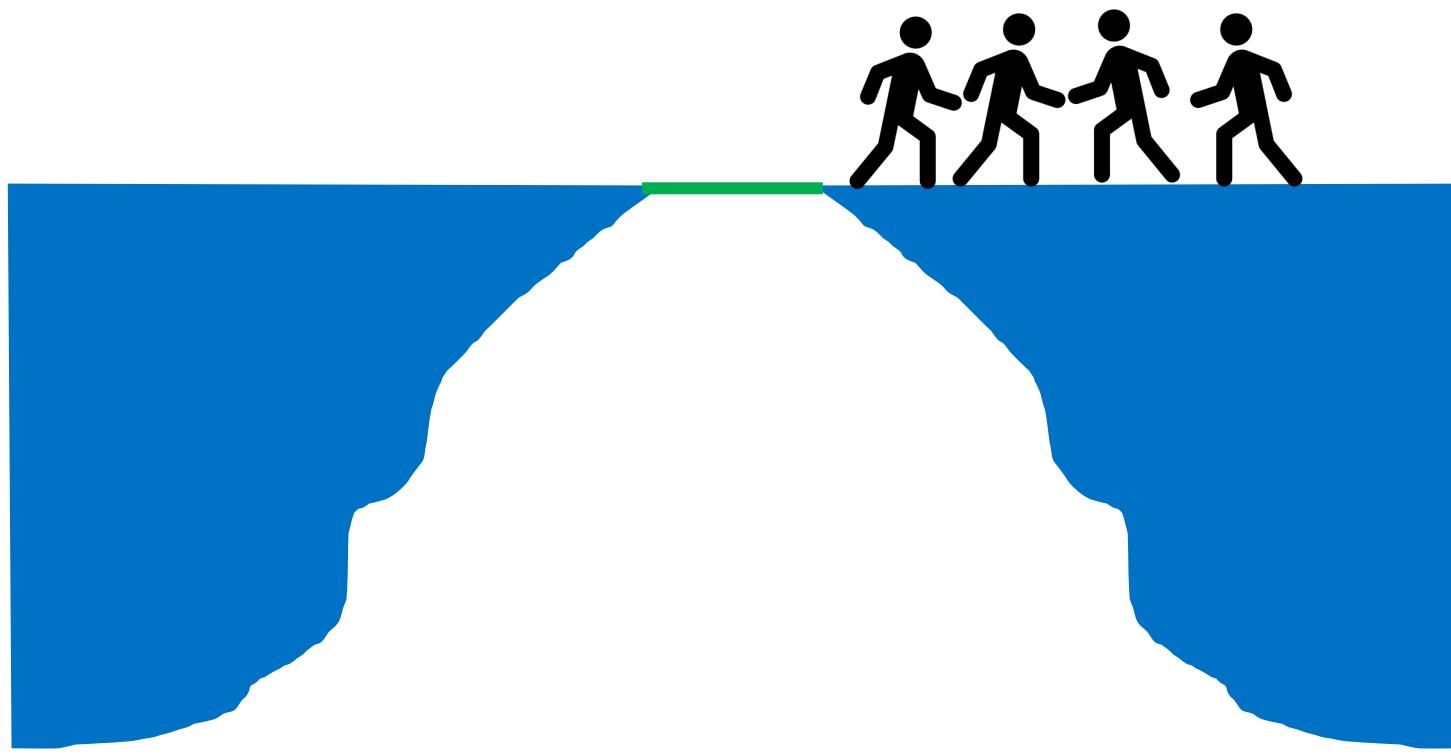
# Avoid Mistakes In The Future



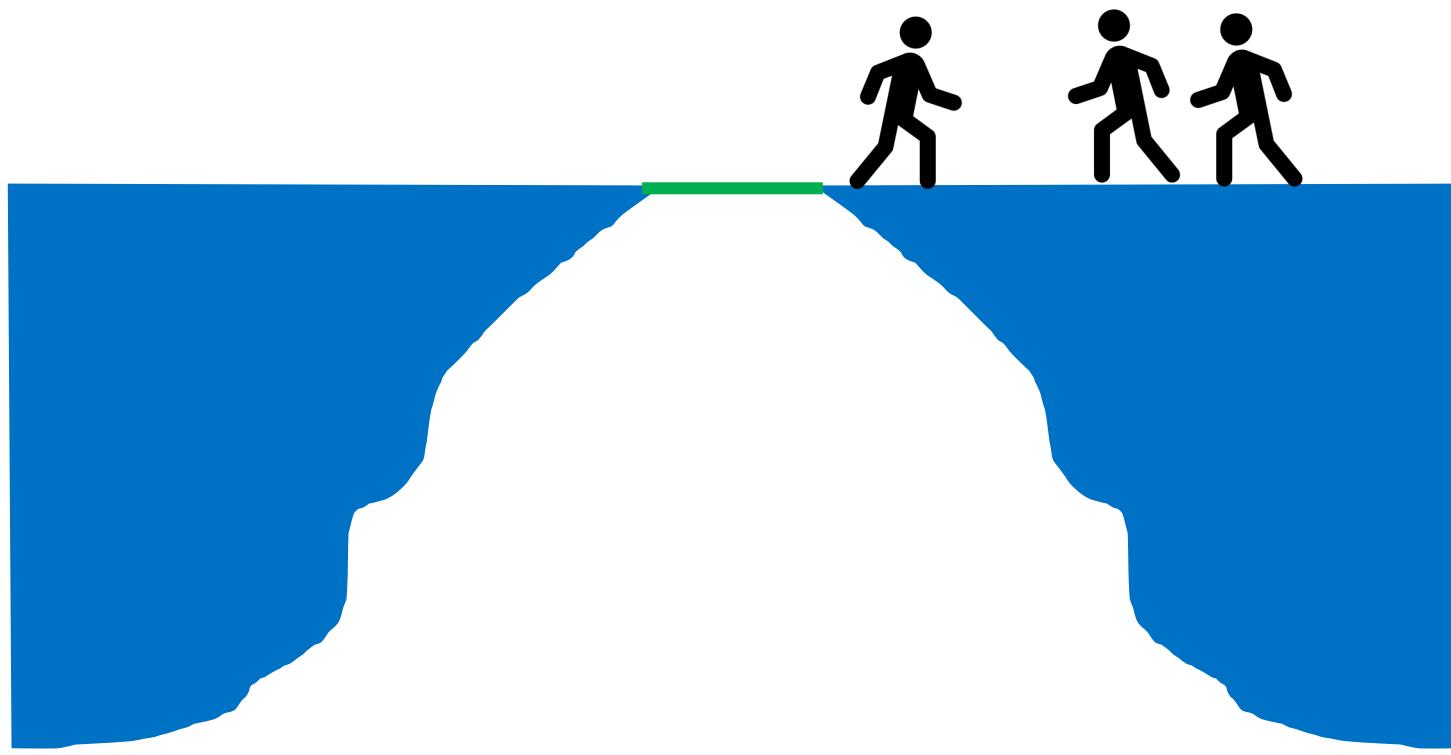
# Avoid Mistakes In The Future



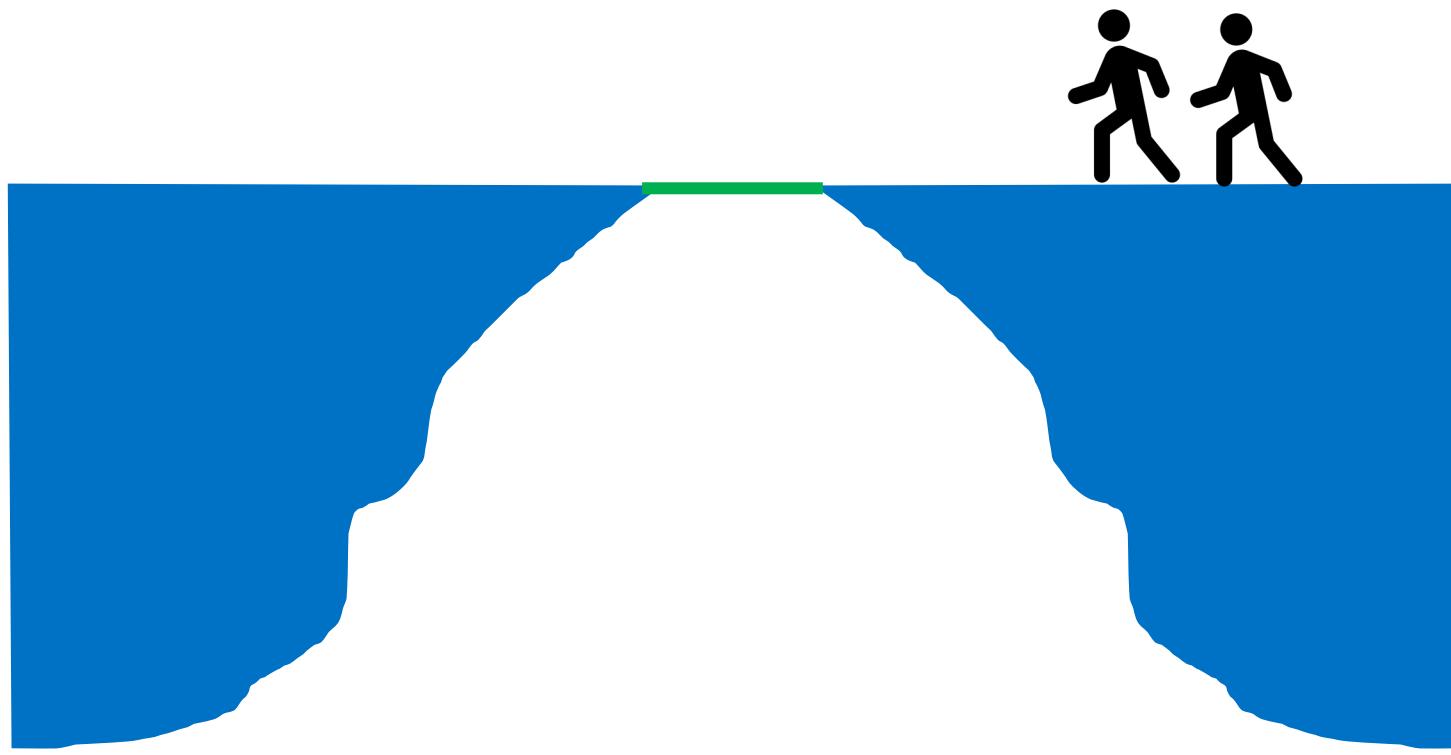
# Avoid Mistakes In The Future



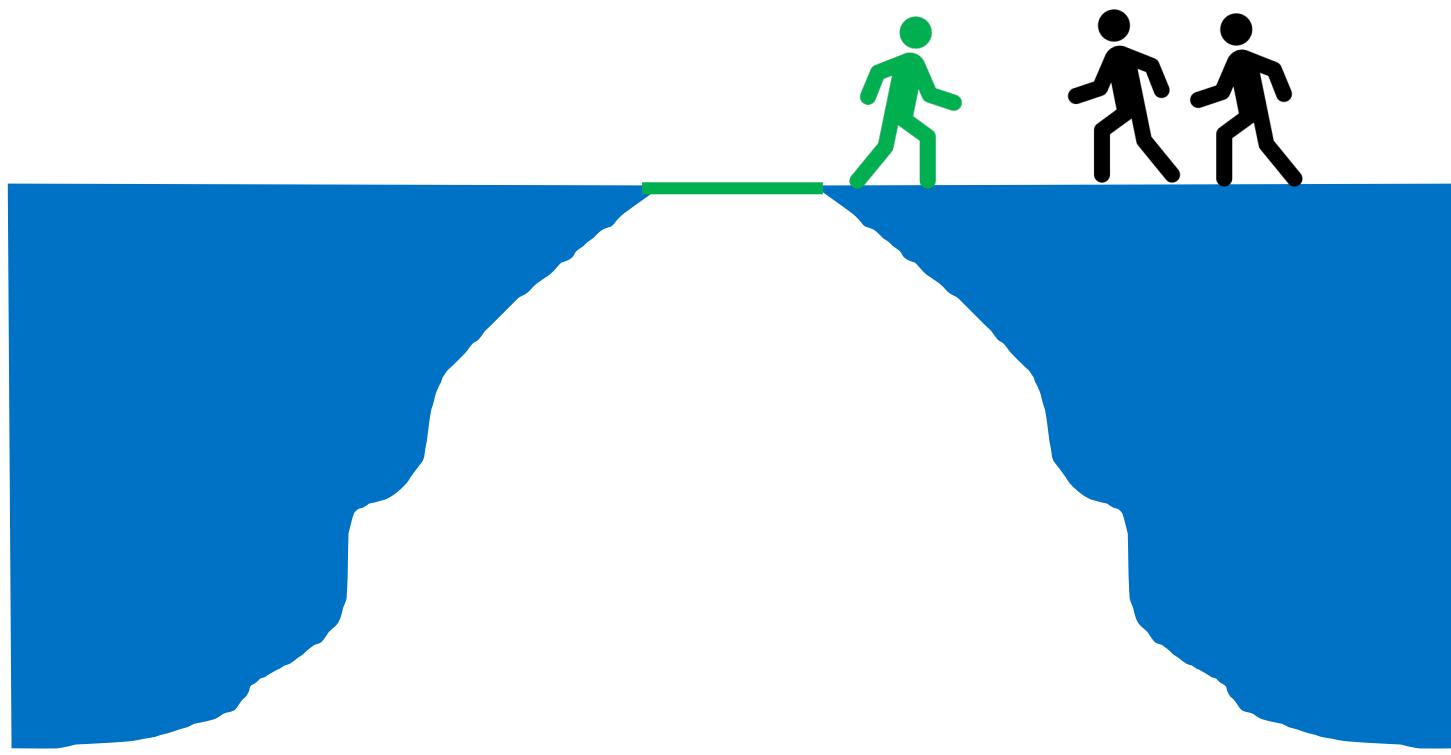
# Avoid Mistakes In The Future



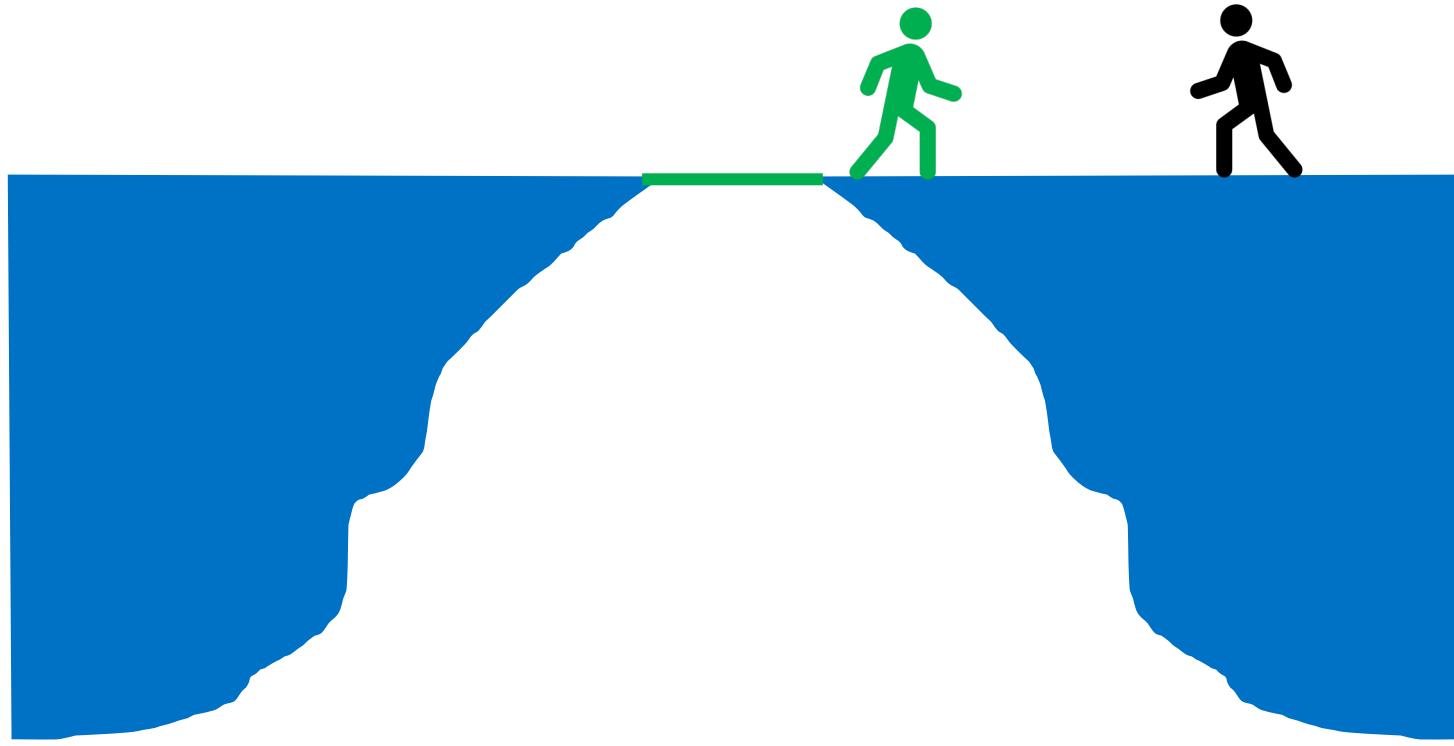
# Avoid Mistakes In The Future



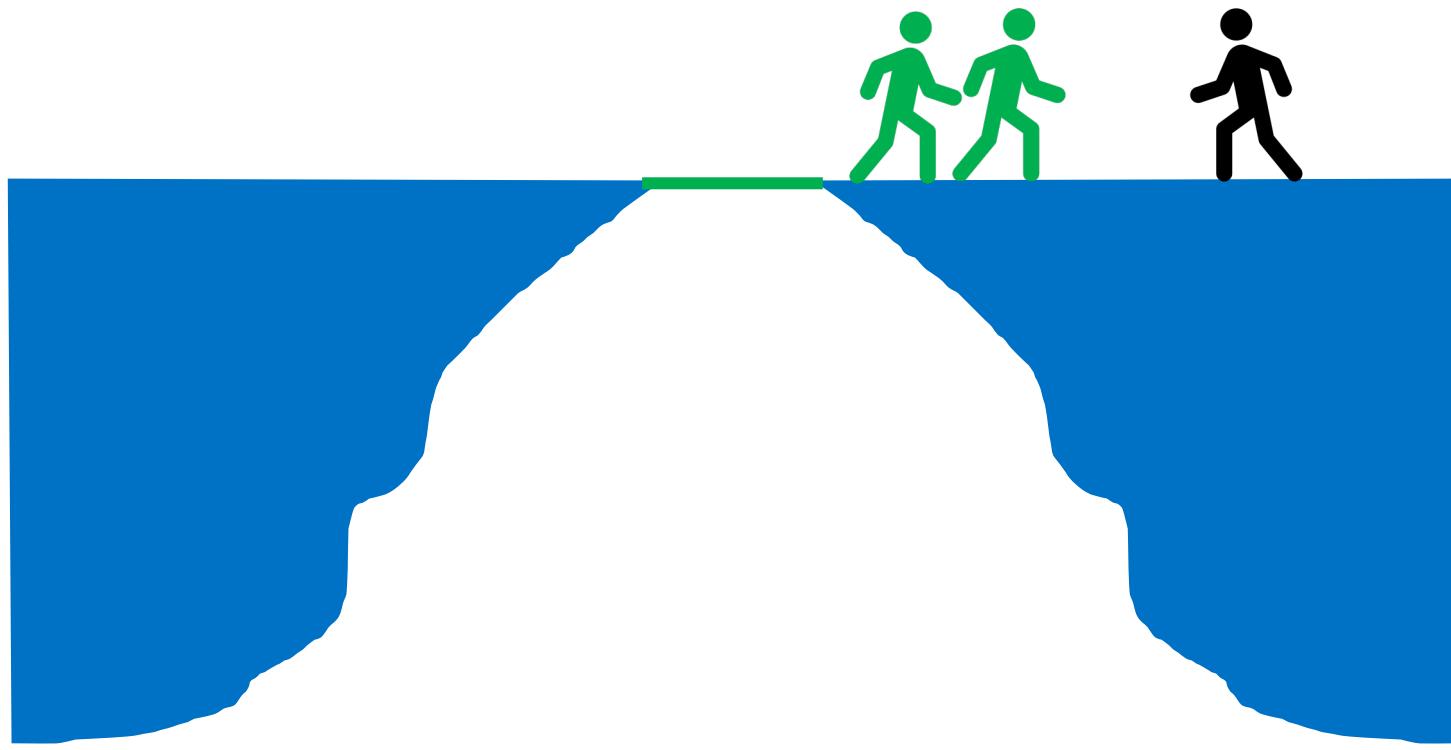
# Avoid Mistakes In The Future



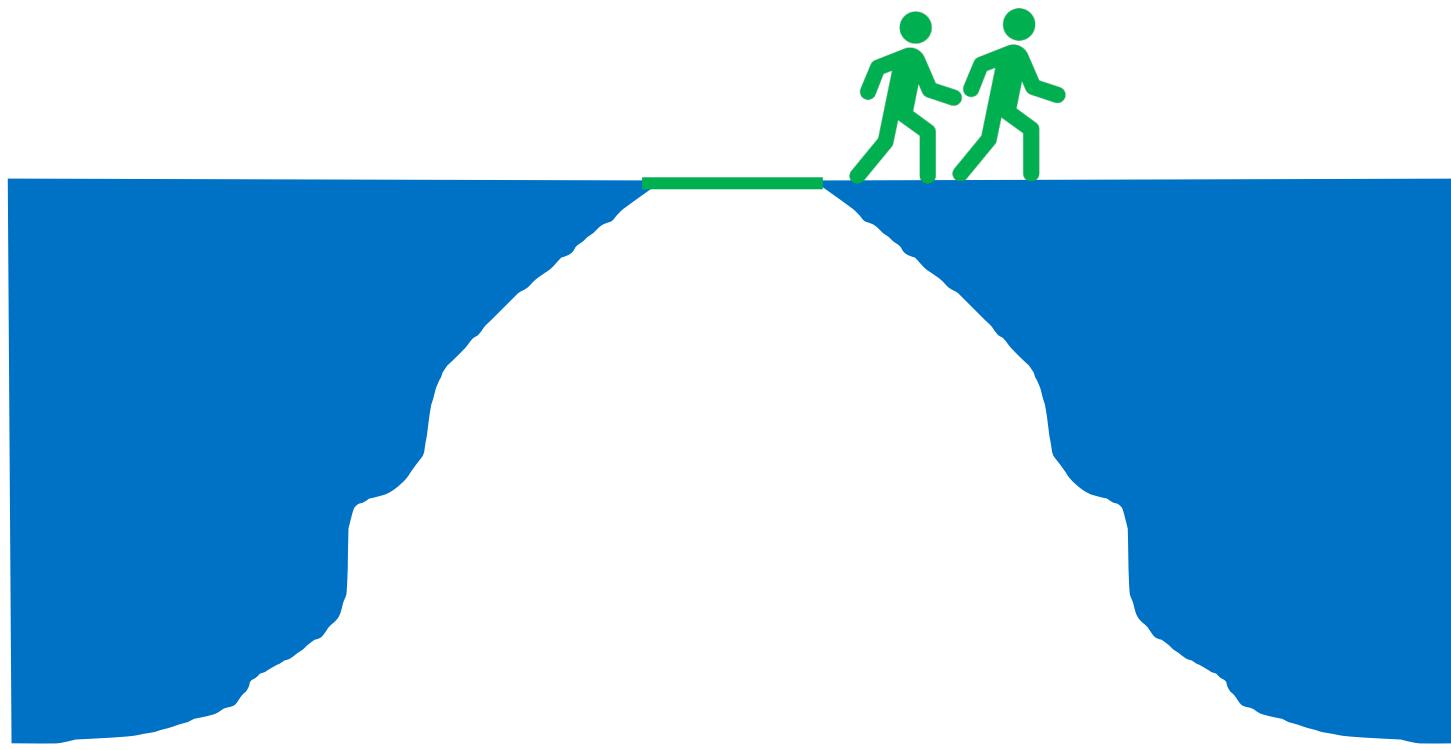
# Avoid Mistakes In The Future



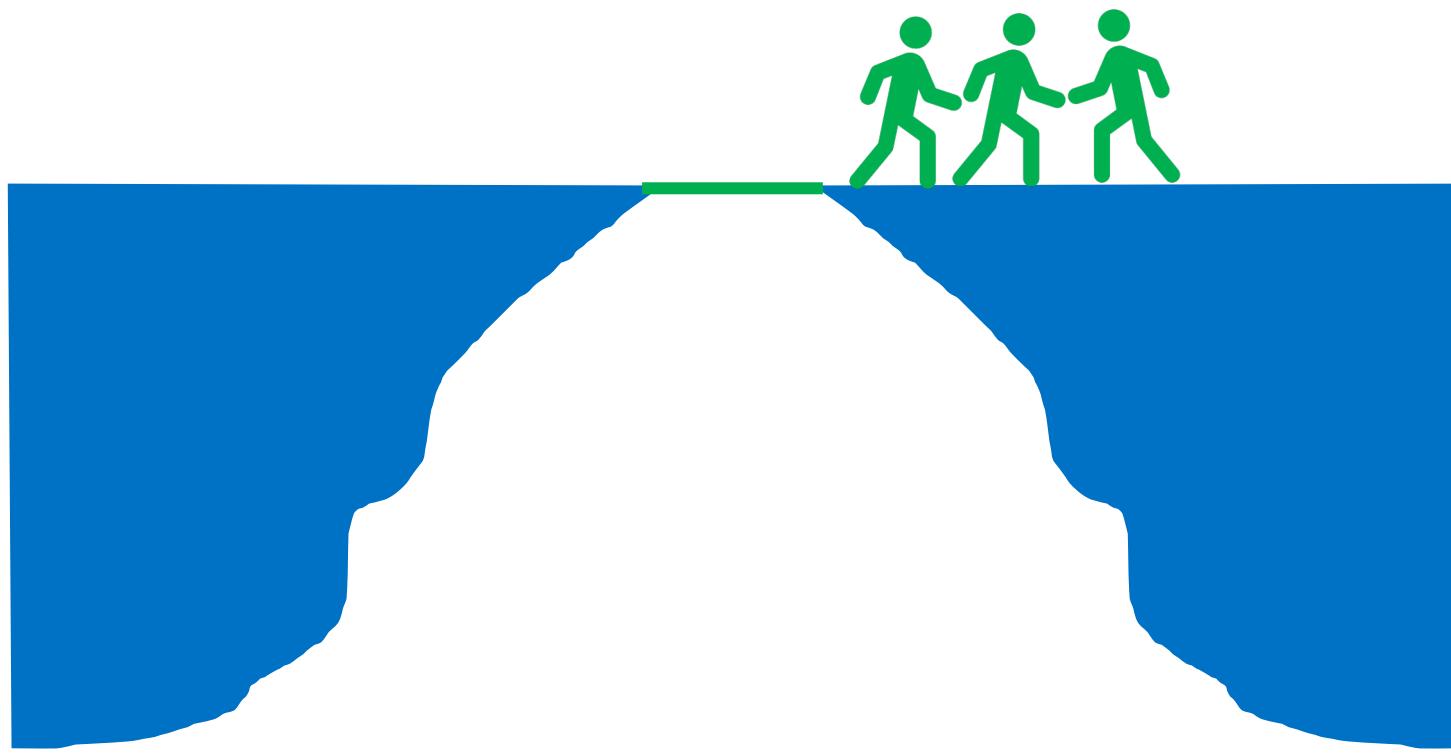
# Avoid Mistakes In The Future



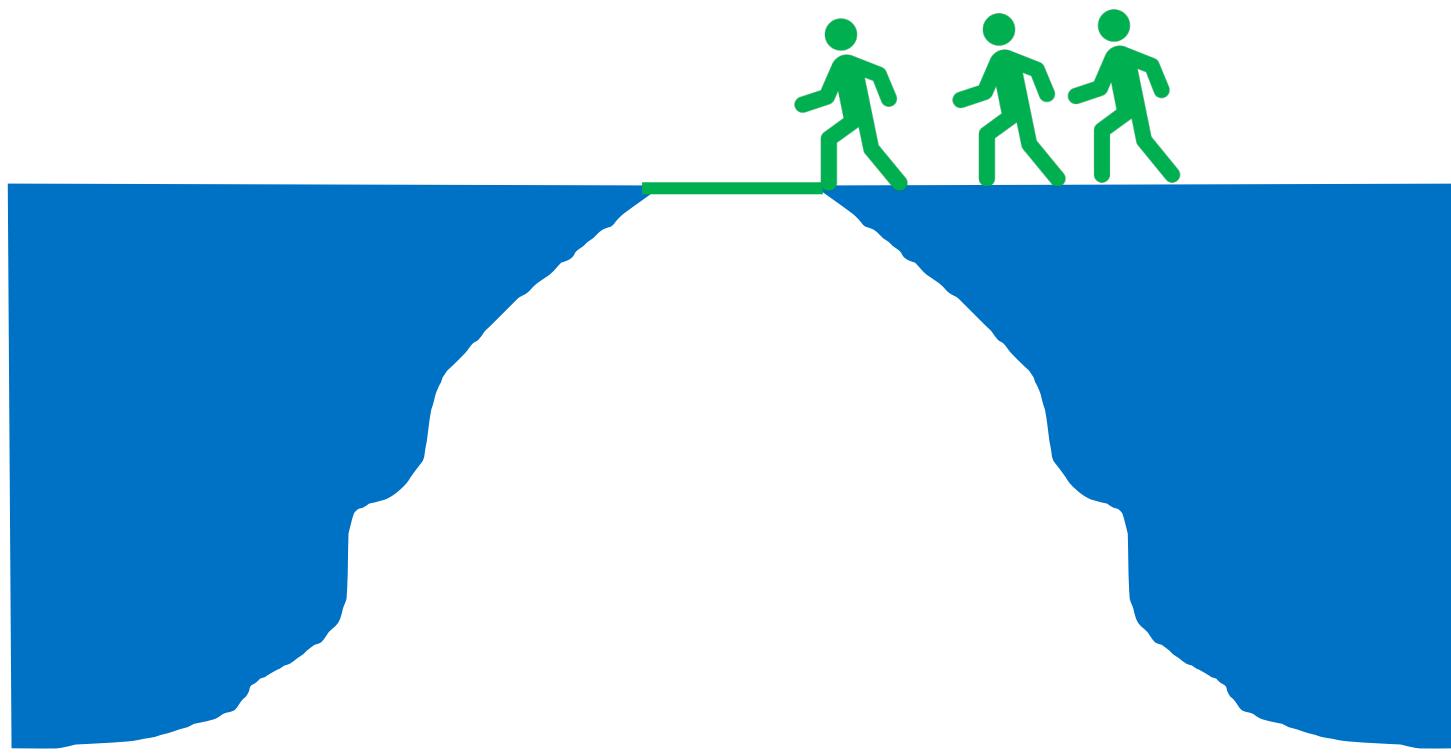
# Avoid Mistakes In The Future



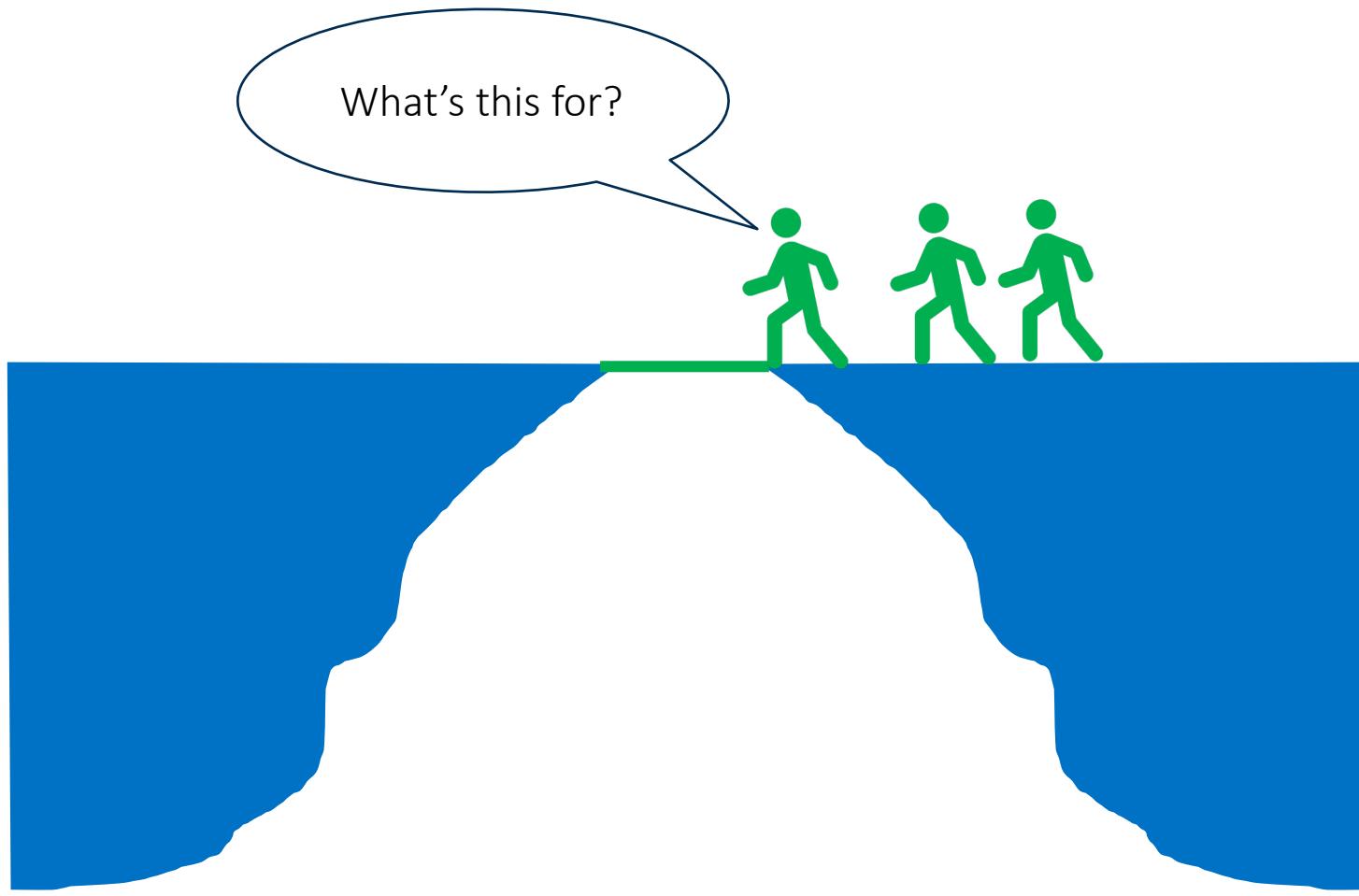
# Avoid Mistakes In The Future



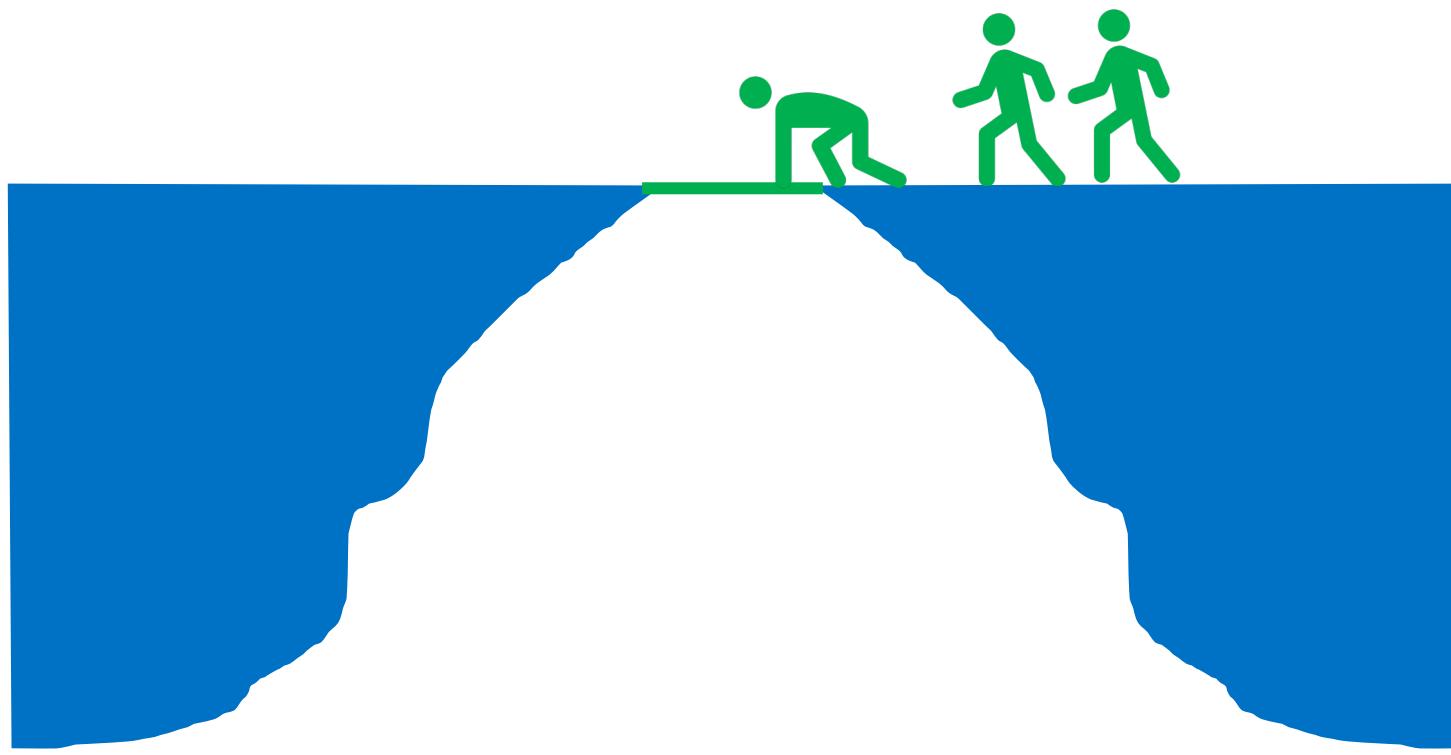
# Avoid Mistakes In The Future



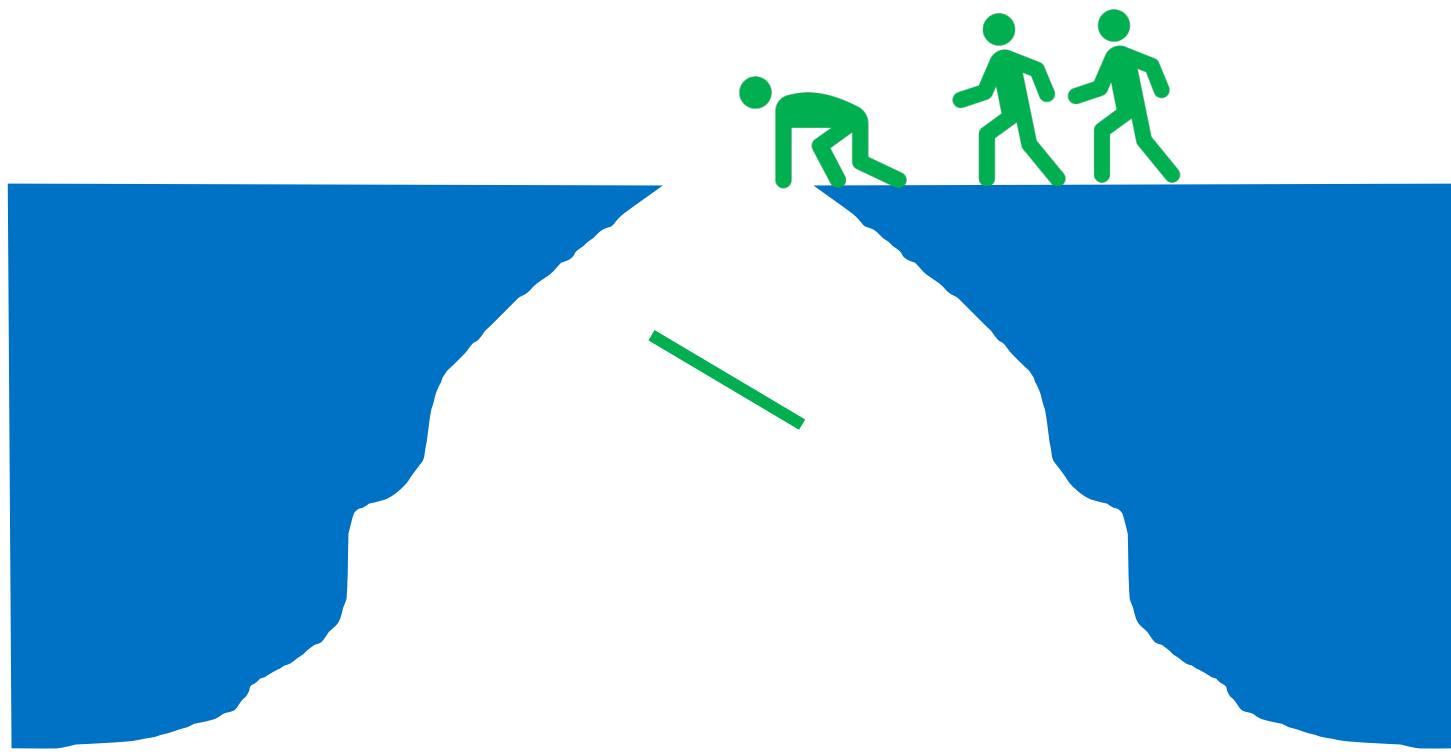
# Avoid Mistakes In The Future



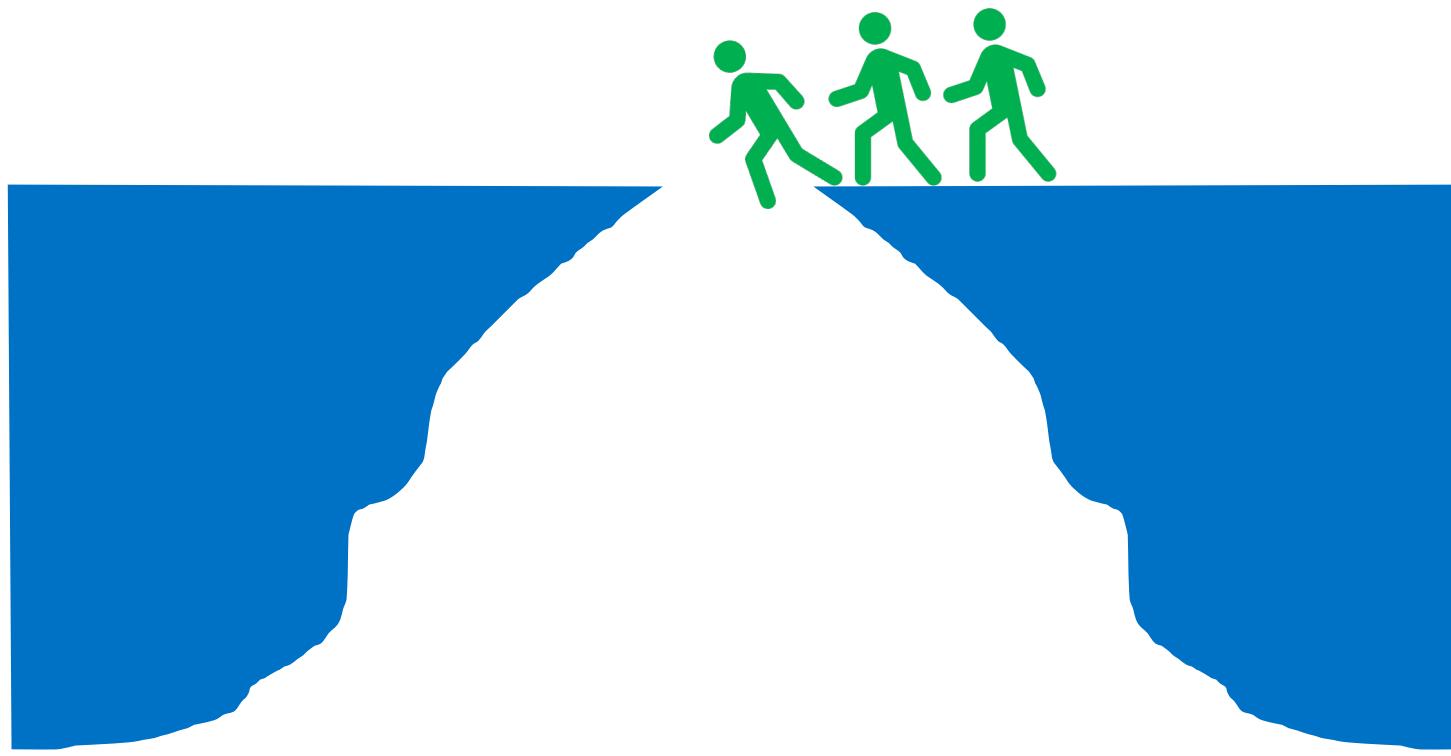
# Avoid Mistakes In The Future



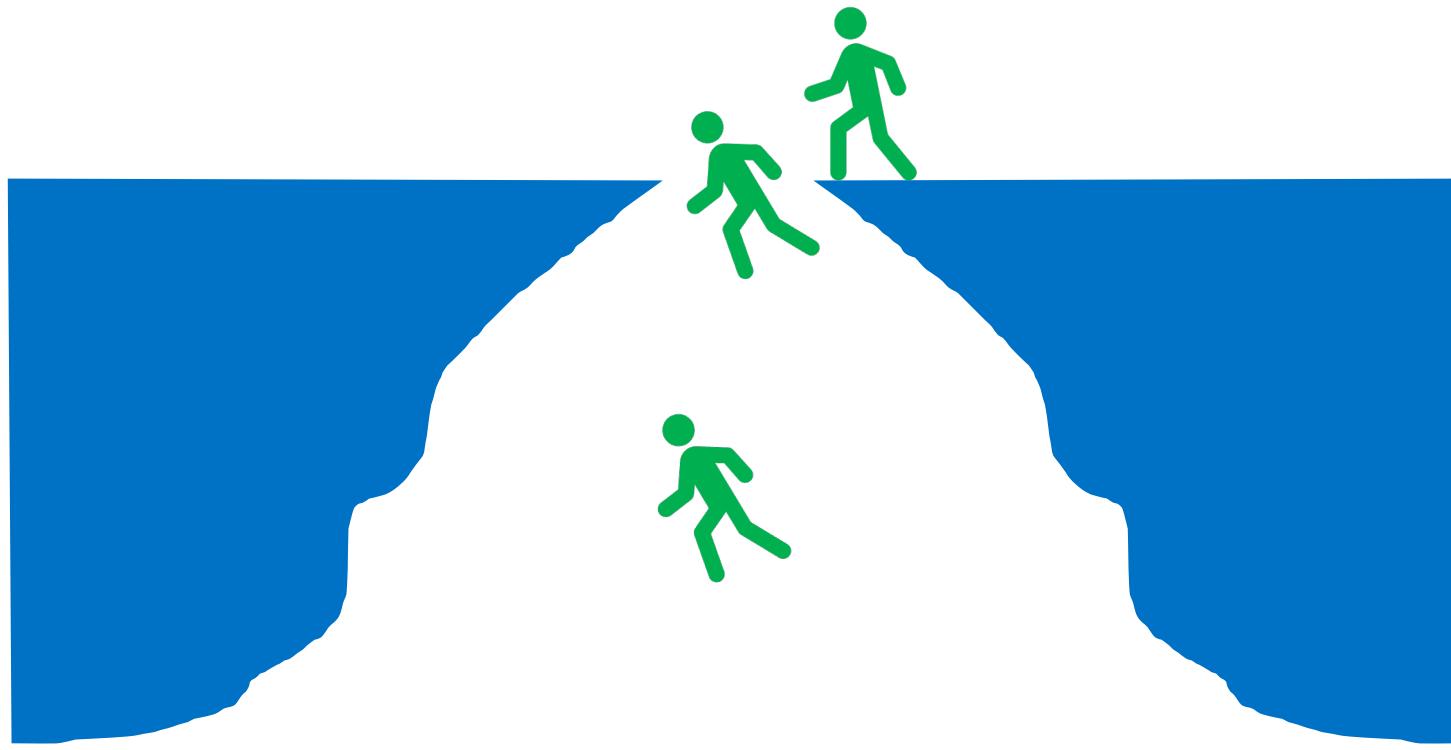
# Avoid Mistakes In The Future



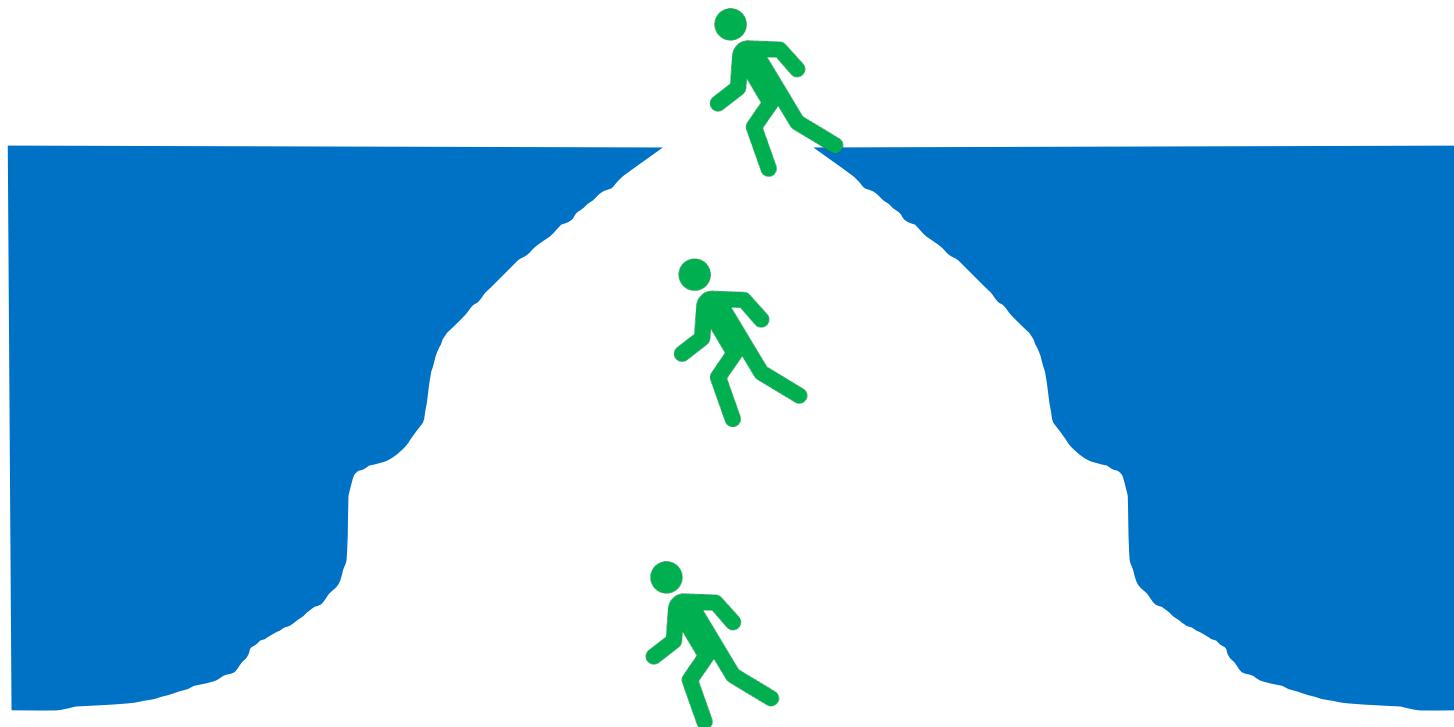
# Avoid Mistakes In The Future



# Avoid Mistakes In The Future



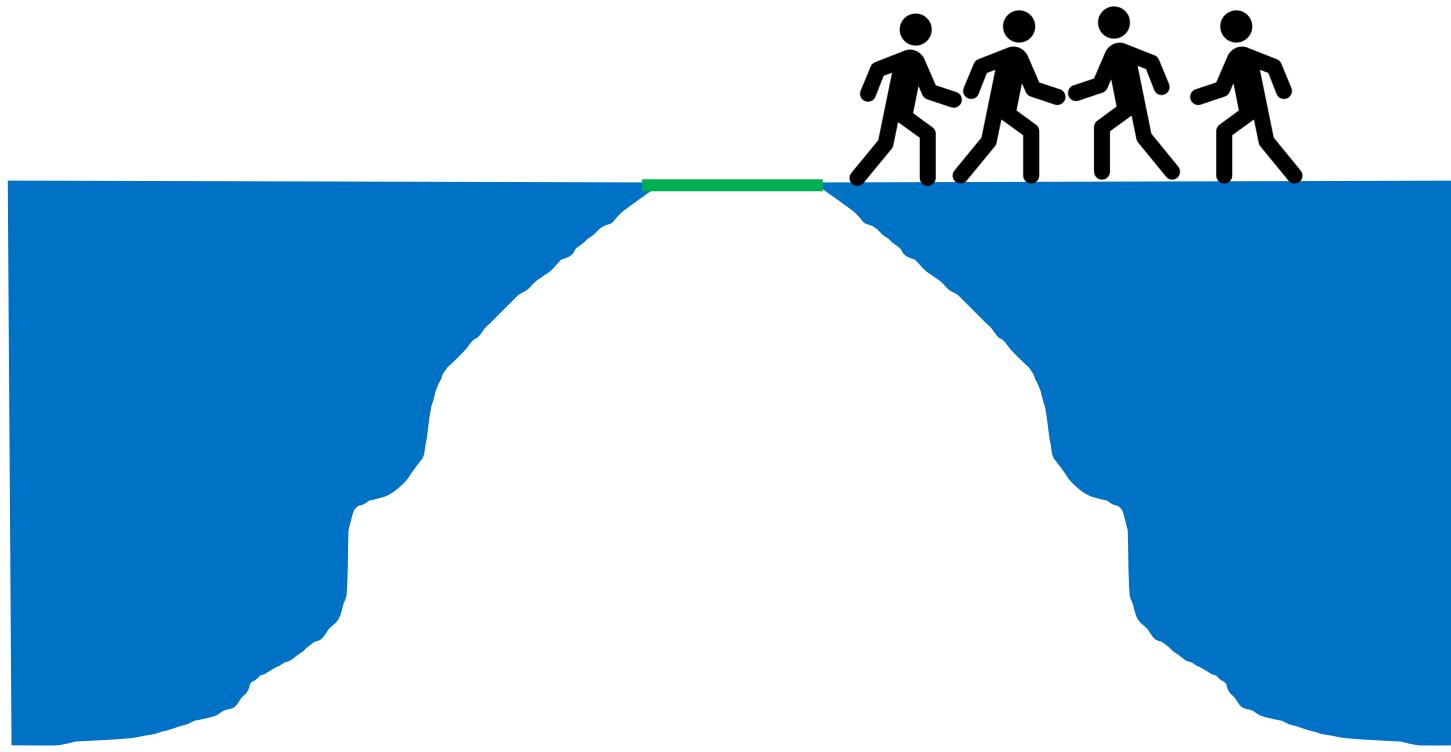
# Avoid Mistakes In The Future



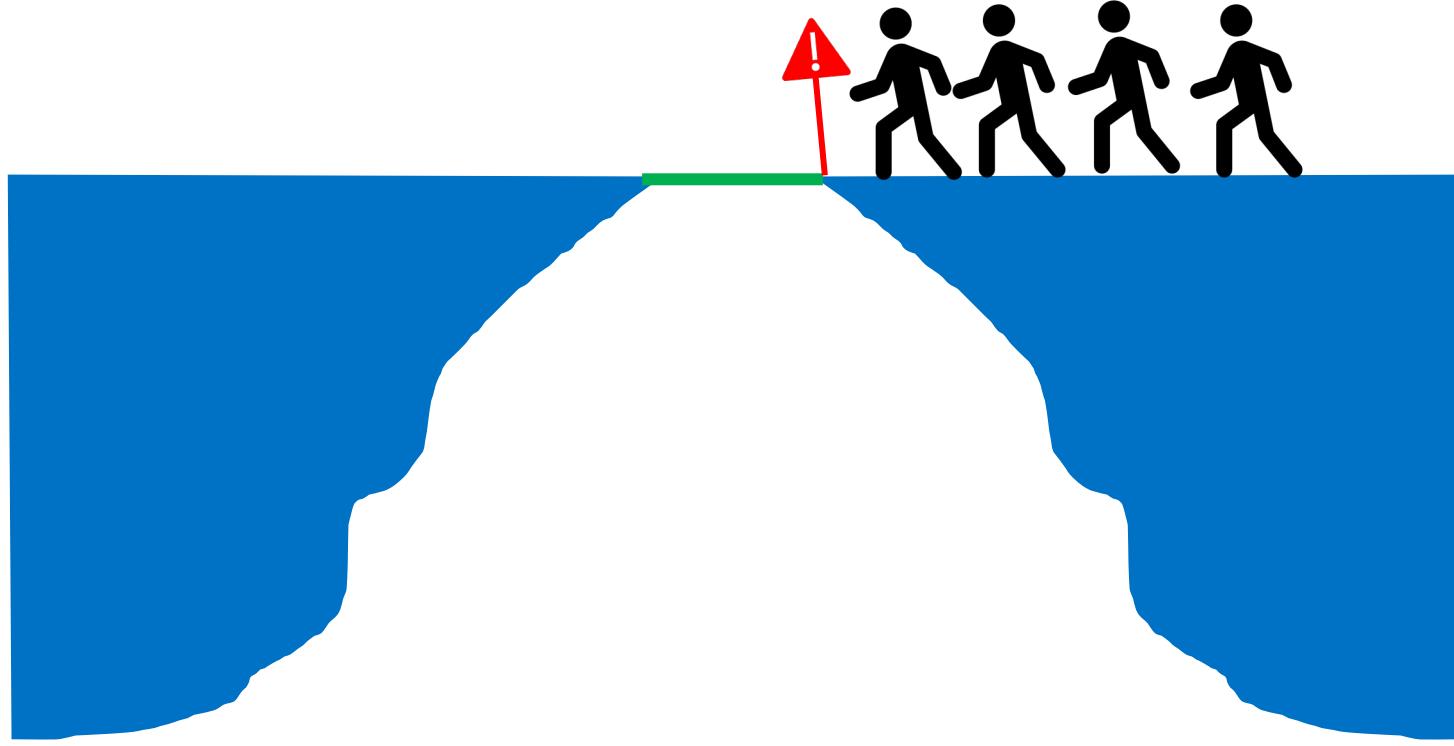
# Avoid Mistakes In The Future



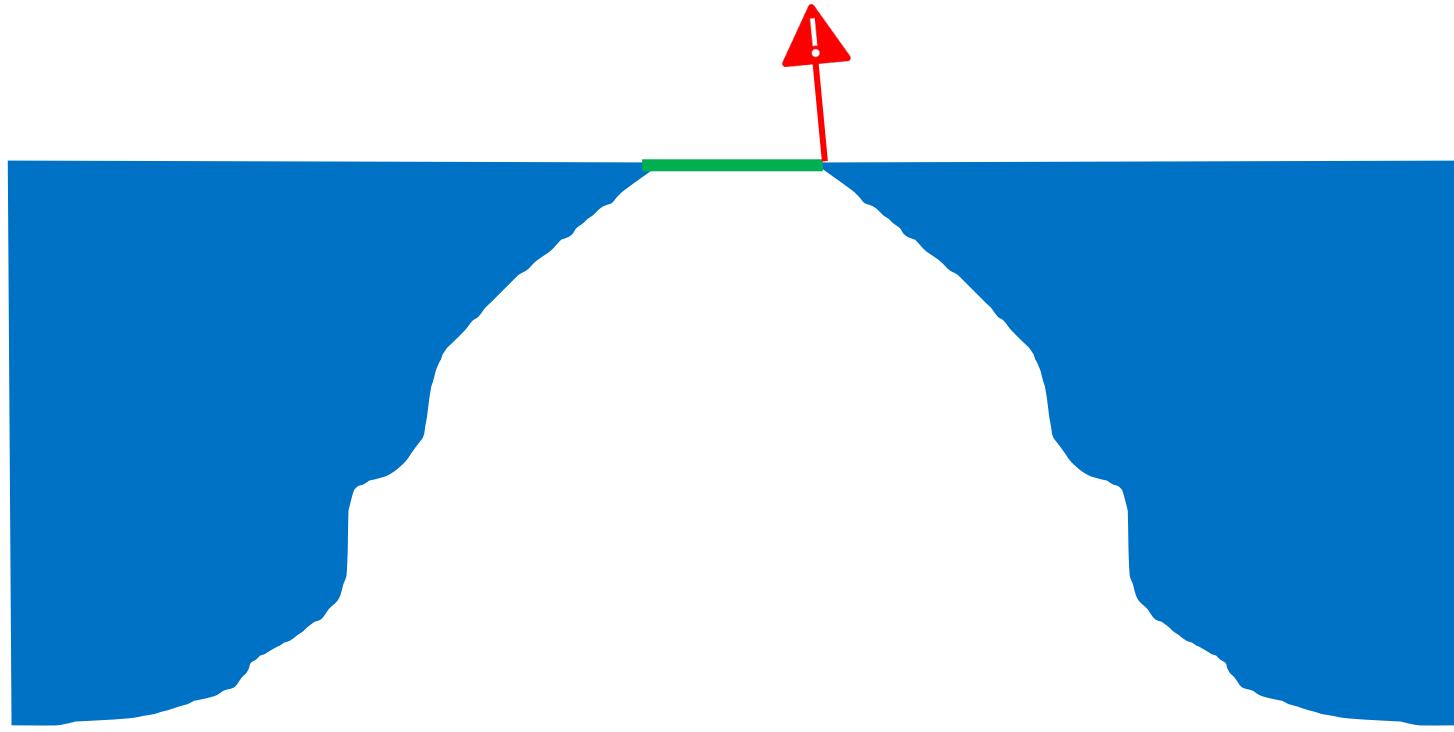
# Avoid Mistakes In The Future



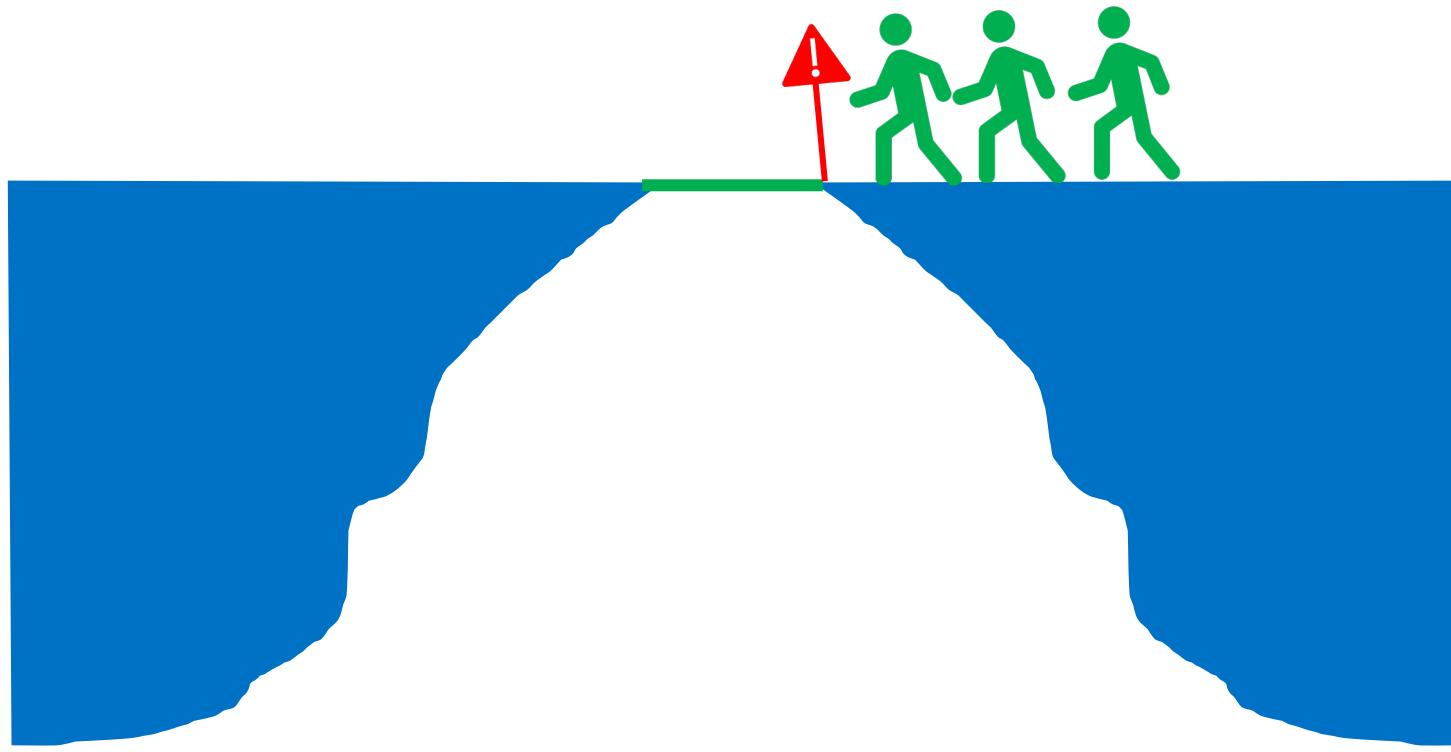
# Avoid Mistakes In The Future



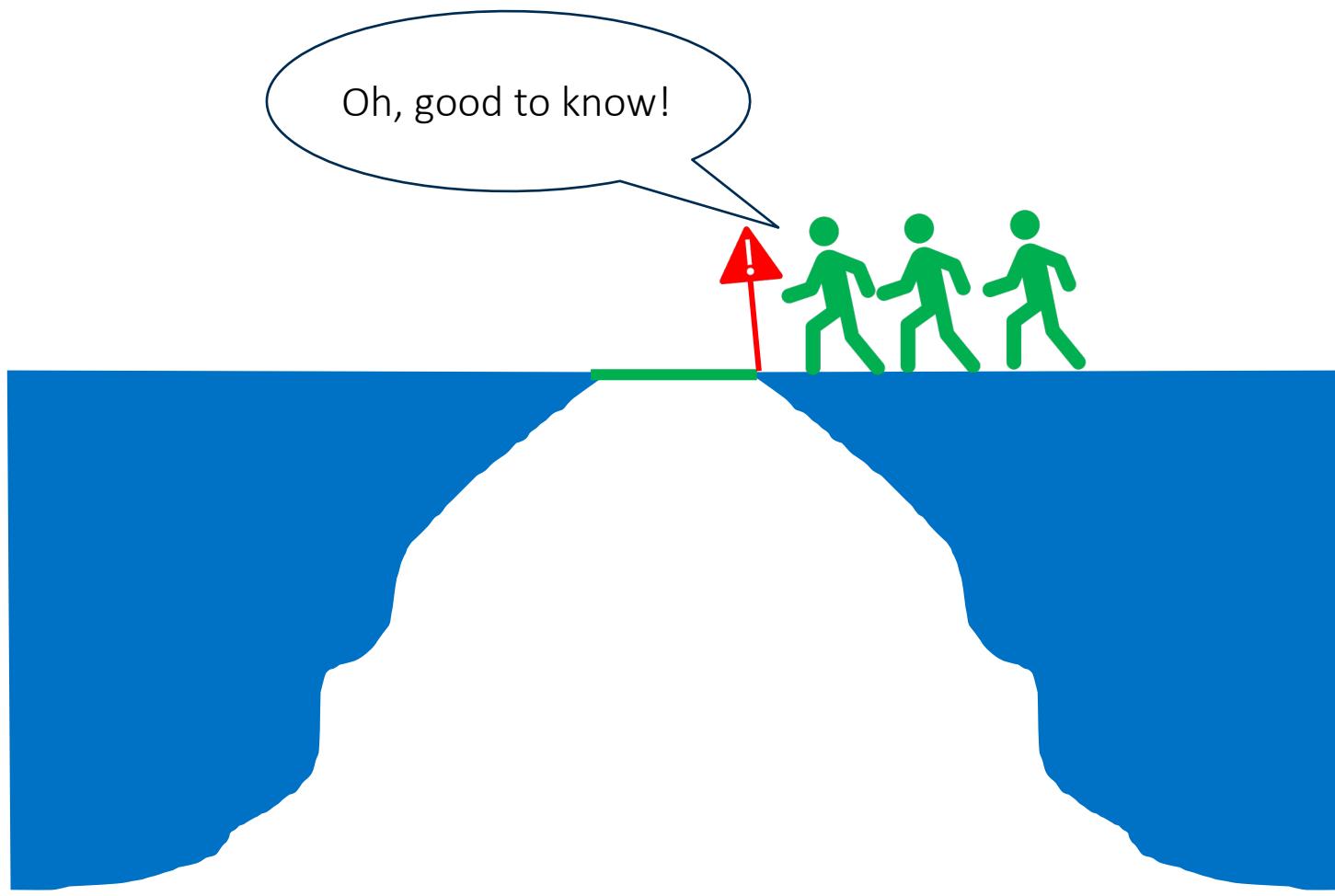
# Avoid Mistakes In The Future



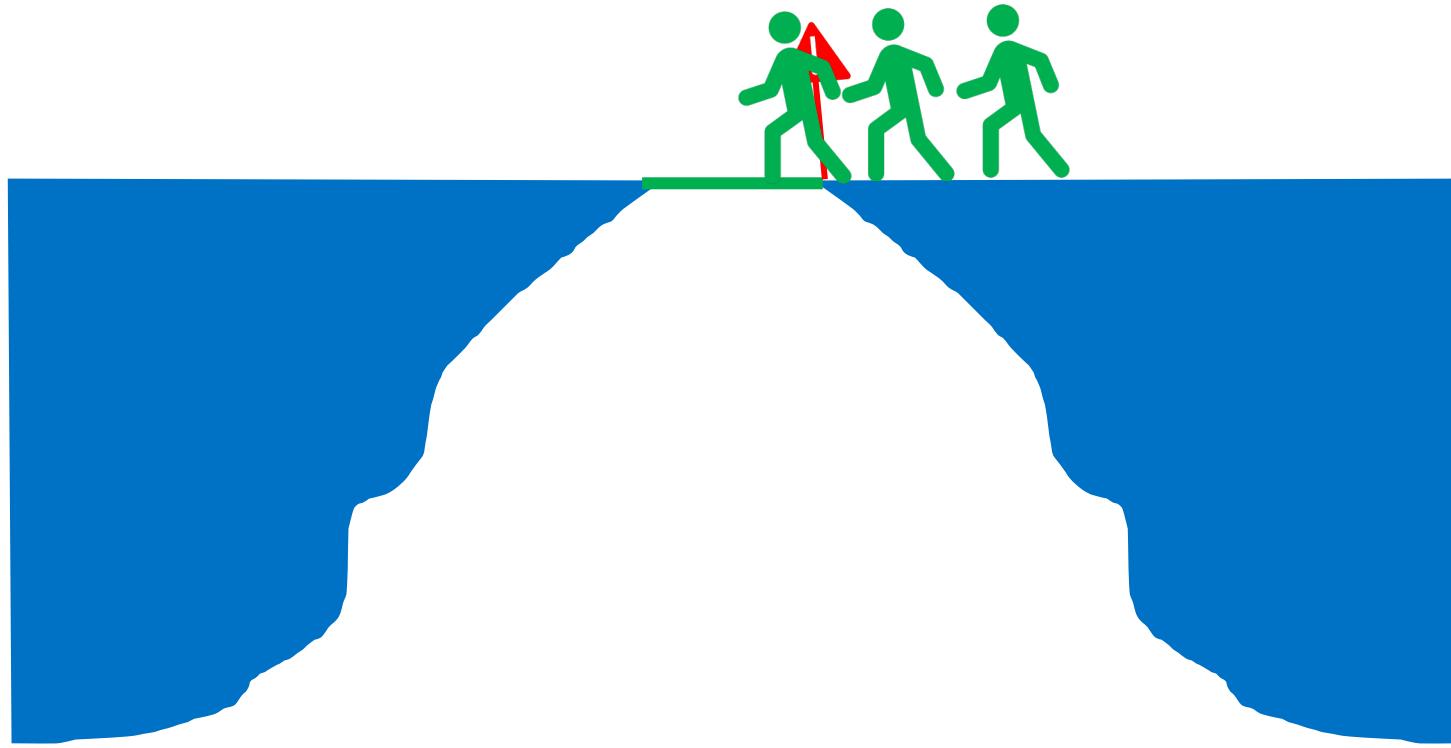
# Avoid Mistakes In The Future



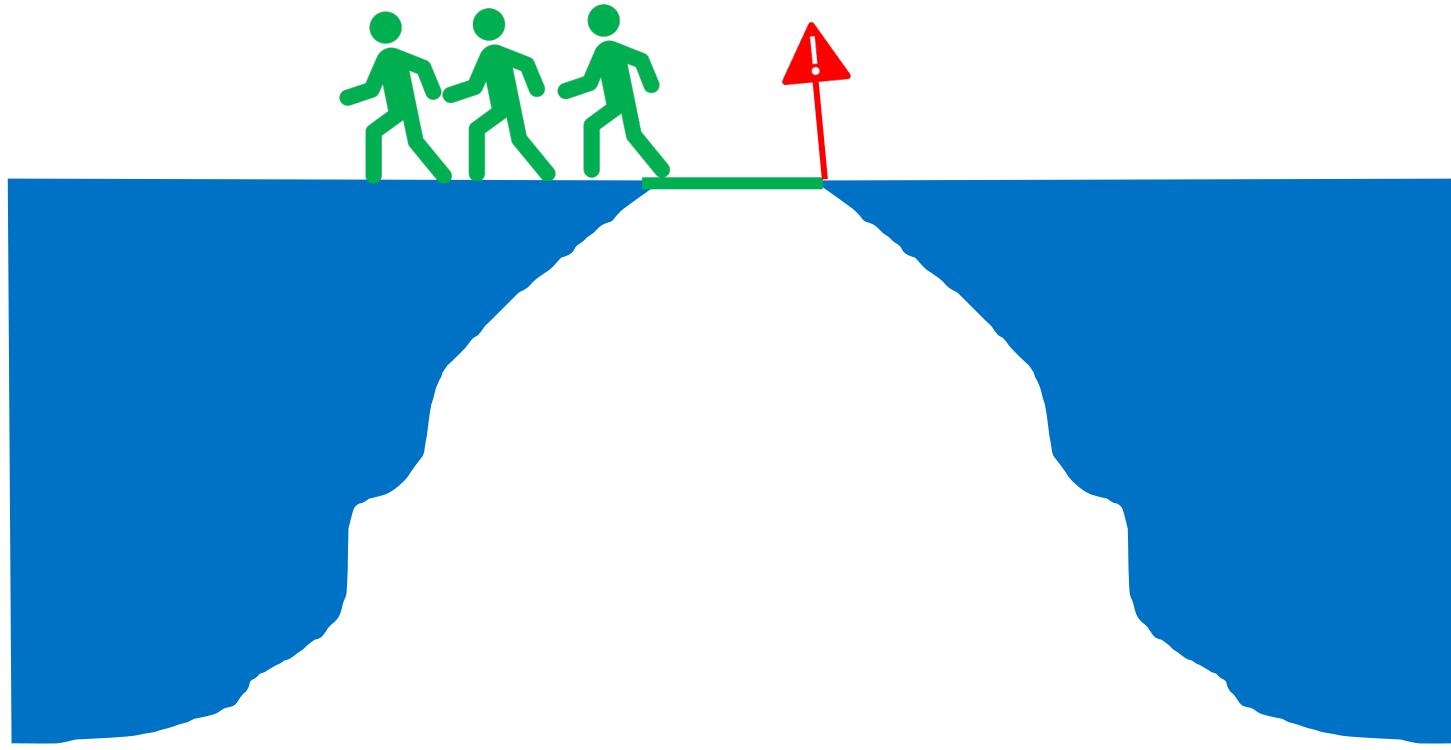
# Avoid Mistakes In The Future



# Avoid Mistakes In The Future



# Avoid Mistakes In The Future



# Architecture Decision Records (ADR)

## ADR.md

**Title:** Avoid Implementing Feature ‘Awaken Balrog’

**Status:** Accepted

**Context:**

In deciding whether to implement the Balrog Awakening feature, we draw inspiration from Gandalf's advice in Moria.

**Decision:**

We will not implement the Balrog Awakening feature to avoid potential catastrophic issues, such as losing our only wizard.

**Consequences:**

Every time we've awaken a Balrog in the past, we've lost our wizard.

# Agile / Iterative Development

Rule 2

# Agile / Iterative Development

- Implement **as little as possible** to solve the problem (**MVP**)
- **Iterate** quickly with **feedback**

# YAGNI (You Aren't Gonna Need It)

Rule 3

# YAGNI



Implement Only  
Current Requirements



Don't Build For  
Future Needs

# KISS (Keep It Simple, Software-developer)

Rule 4

MORE POWER, LESS PARTS

SPACEX



V1: 185t



V2: 230t



TRAILHEAD  
TECHNOLOGY PARTNERS

# KISS



Less Surface Area  
For Failures



Easier to  
Maintain

# DRY (Don't Repeat Yourself)

Rule 5

# DRY

- Reduce the amount of duplicate code written.
- This can save time and energy,
- Helps developers focus on more important problems.
- Can also make software more maintainable, readable, and consistent

# Principle of Least Astonishment (POLA)

Rule 6

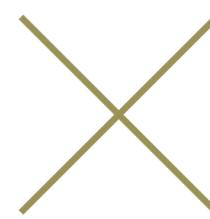
# Principle of Least Astonishment

**"A component of a system should behave in a way that most users will expect it to behave, and therefore not astonish or surprise users."**

# Principle of Least Astonishment (POLA)



Reduces  
Cognitive Load



Reduces  
Mistakes



Easier to  
Learn

# Pull Reqs / Code Reviews

Rule 7

# Pull Reqs / Code Reviews



Detecting Patterns  
of Over-  
Engineering



Catching  
Unnecessary  
Features Early



Ensuring  
Adherence to Best  
Practices



Encouraging  
Incremental  
Development

# Typical Code Review Checklist

- Bug Check
- Acceptance Criteria Check
- Code Standards Check
- Clarify Check
- Performance Check
- Documentation Check

# Typical Code Review Checklist

- Bug Check
  - Acceptance Criteria Check
  - Code Standards Check
  - Clarify Check
  - Performance Check
  - Documentation Check
- 
- ✓ Simplicity Check
  - ✓ YAGNI Check
  - ✓ KISS Check
  - ✓ Premature Optimization Check

# Feedback Loops

Rule 8



# Feedback Loops



Aligning Development  
with User Needs

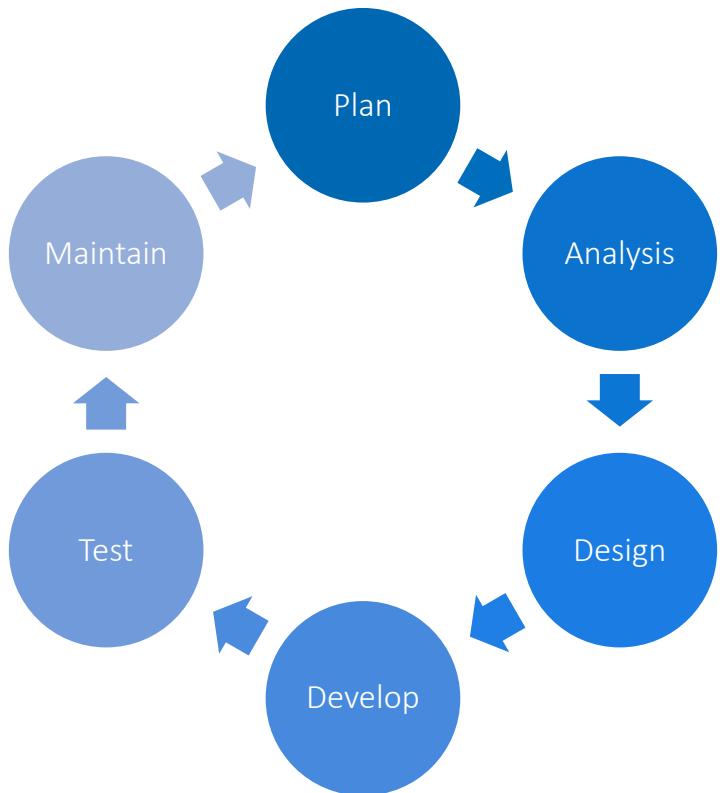


Encouraging Iterative  
Development

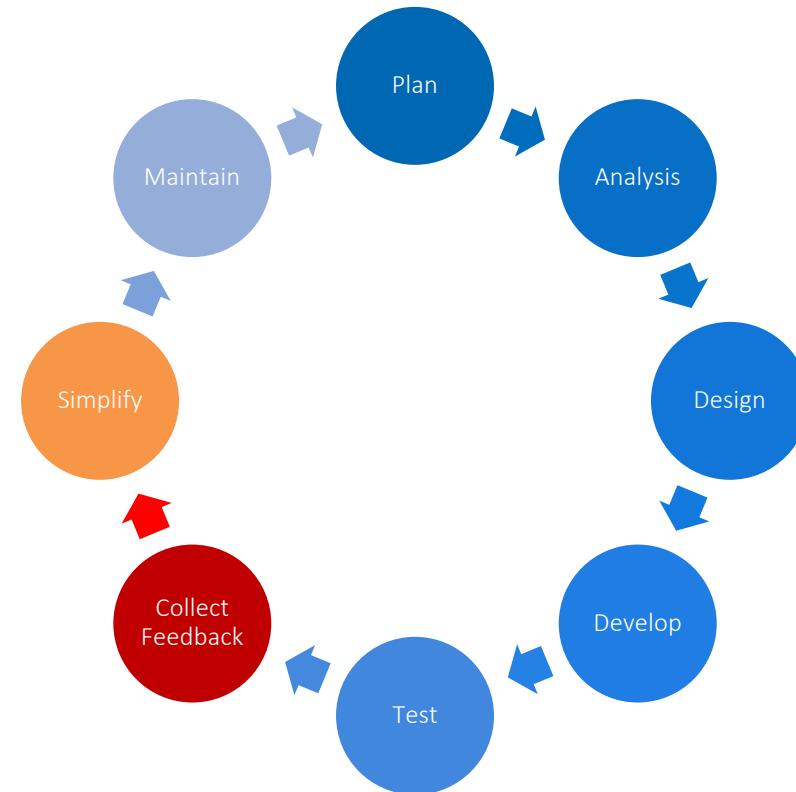
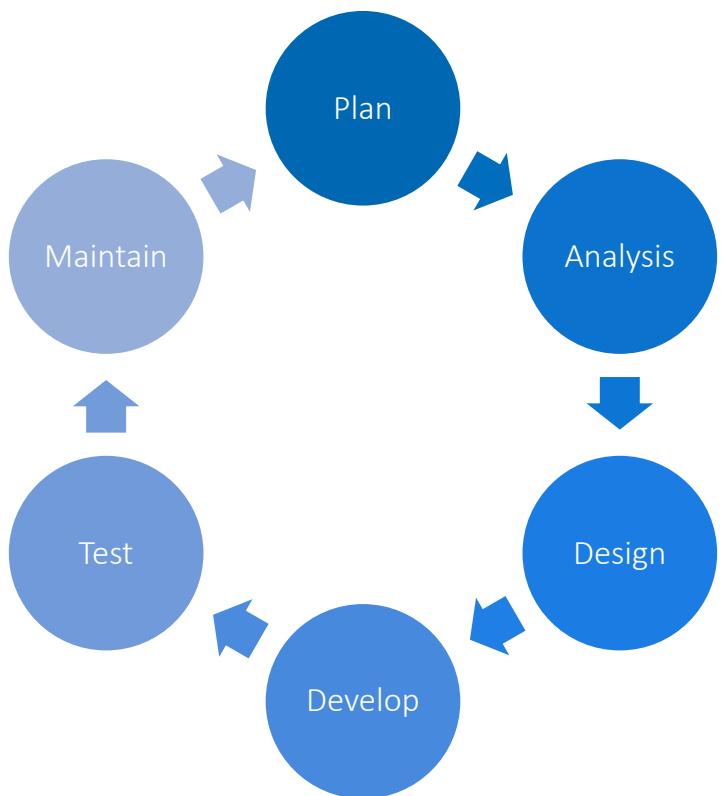


Detecting and correcting  
over-engineering early

# Feedback Loops



# Feedback Loops



# Adopt Proven Solutions

Rule 9

# Adopt Proven Solutions



# Adapting Proven Solutions



More predictable  
outcomes



Reduces unknowns



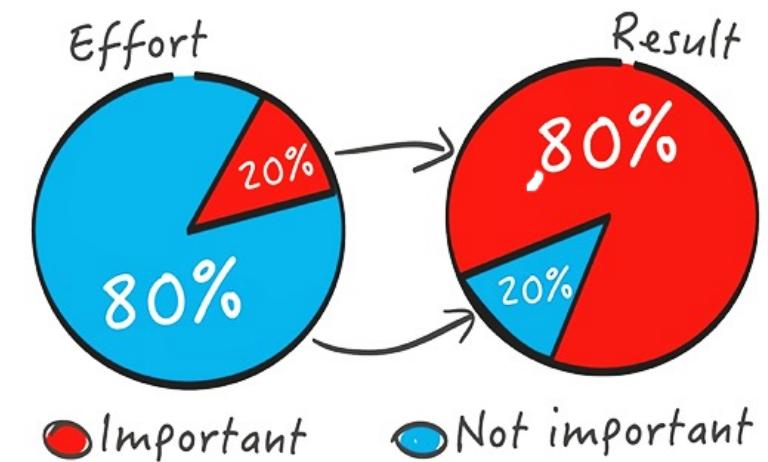
Manages risk

# 80/20 Rule (Pareto Principle)

Rule 10

# 80/20 Rule

“Roughly 80% of consequences come from 20% of causes.”

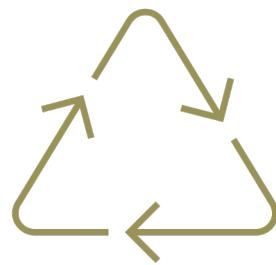


SOURCE <https://www.sreedep.com/the-pareto-principle-explained/>

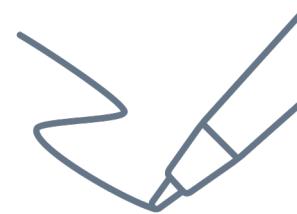
# 80/20 Rule Allows



Focusing on  
high-impact features



Efficient  
use of resources



Avoid  
perfectionism

# Summing Up

1. Over-engineering is **common** but **costly**
2. **Know the signs** of over-engineering
3. Implement the **rules to avoid** over-engineering



# Thanks! Questions?

## Jonathan "J." Tower

🏆 Microsoft MVP in .NET

✉️ jtower@trailheadtechnology.com

🌐 trailheadtechnology.com/blog

🐦 jtowermi

linkedin jtower

[github.com/trailheadtechnology/over-engineering](https://github.com/trailheadtechnology/over-engineering)

**FREE  
CONSULTATION**



[bit.ly/th-offer](http://bit.ly/th-offer)