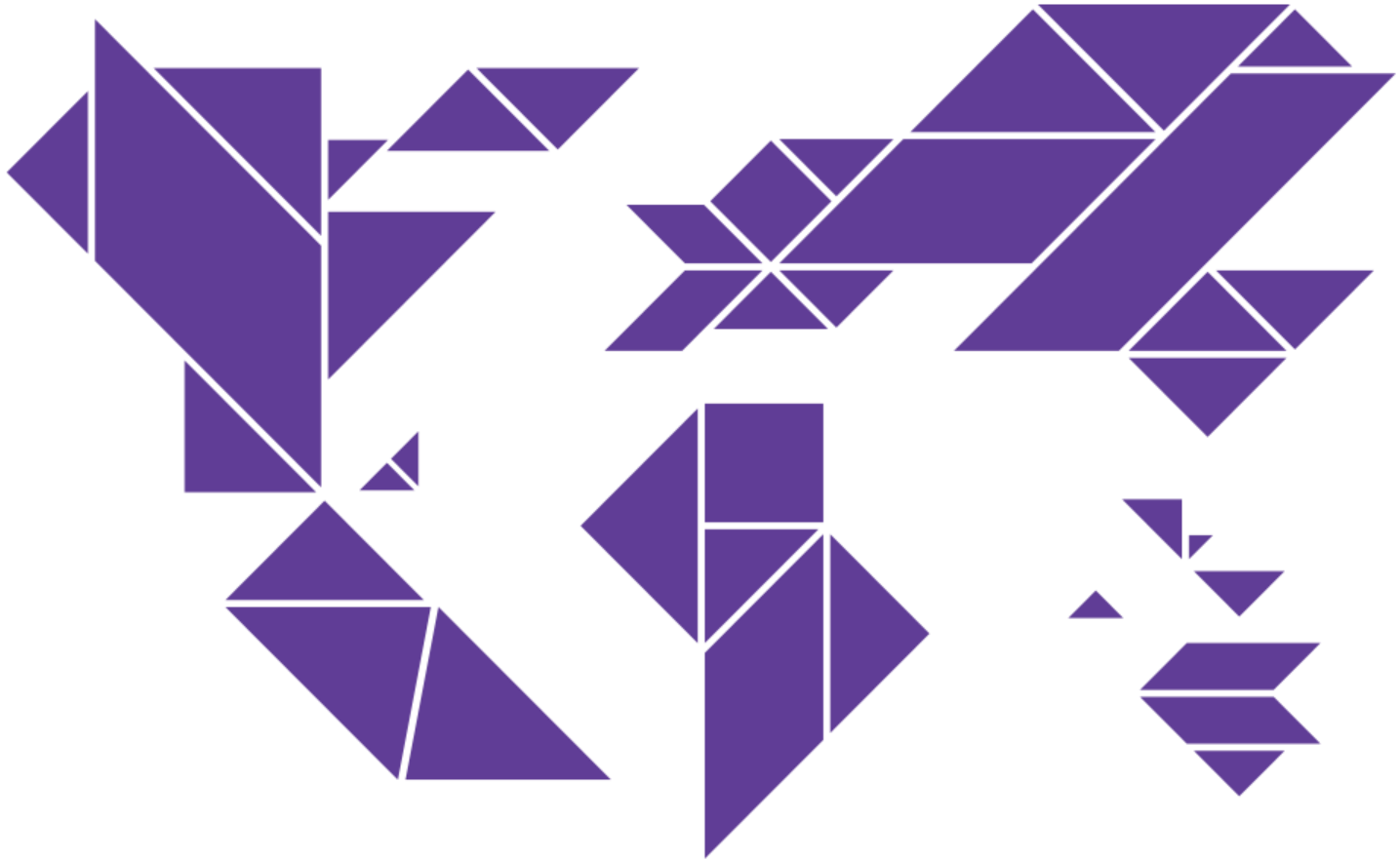


# Old New: Best Practices For Modernizing Legacy Applications With Microservices

terazo
Building Platforms That Scale

# Introductions

Kevin Israel
Richmond Virginia
19 years in Information Technology
Background is in software engineering and architecture

Founder/organizer of RVA Software Development User Group
Market Technical Director at Terazo
Focused on Client Delivery and Technical Solutions
Terazo – Software Engineering (with an API focus), DevOps and Data Engineering

terazo
Building Platforms That Scale

# What we will cover

- Why we are here
- Legacy application lifecycle
- Peeling the onion (it will make you cry)
- Types of monoliths
- Why microservices
- Reaching the turning point
- Best Practices
  - Decompose using iterative process
  - Get a product owner
  - Containerize
  - Rough design up front
  - Follow good API development patterns
  - Develop iteratively
- Best Practices continued
  - Decompose with roadmap in mind
  - Implement API gateway
  - Work towards an event driven architecture
  - Write good tests

# Why are we here?

The reality is that many organizations have – and some struggle with – legacy applications.

Typically the legacy application was:

- Built on older technology stacks.
- Deployed to production with technical debt.
- Designed for now, with no vision of the future.
- Not designed for functional testing with automated tooling.
- Not built in a loosely coupled way so flexibility is limited, change is difficult, scalability is limited.

# Why are we here?

Typically the legacy application is:

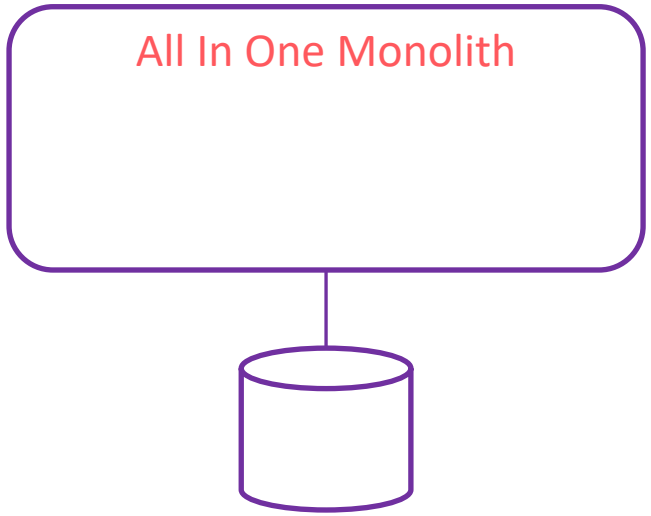
- critical to the operations of the company
- generating revenue for the company
- hard to make changes to
- not documented well
- nearing it's end of life

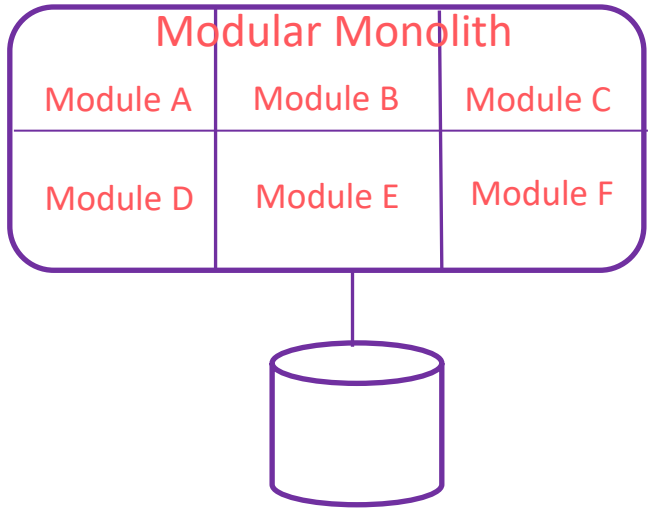
# The lifecycle of a legacy application

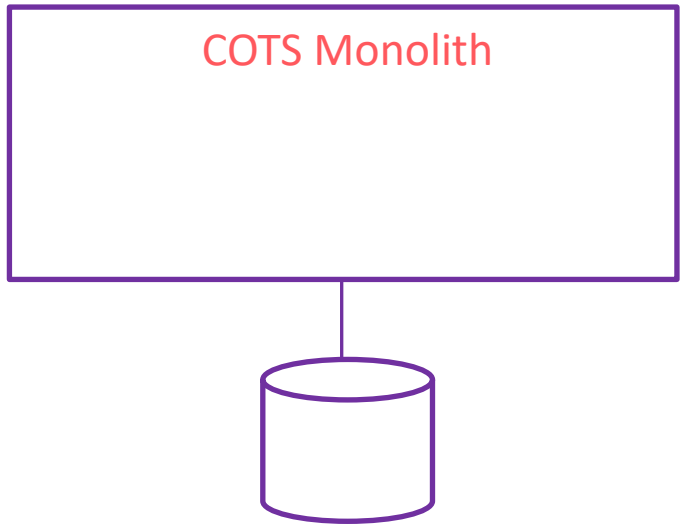
# Peeling the onion

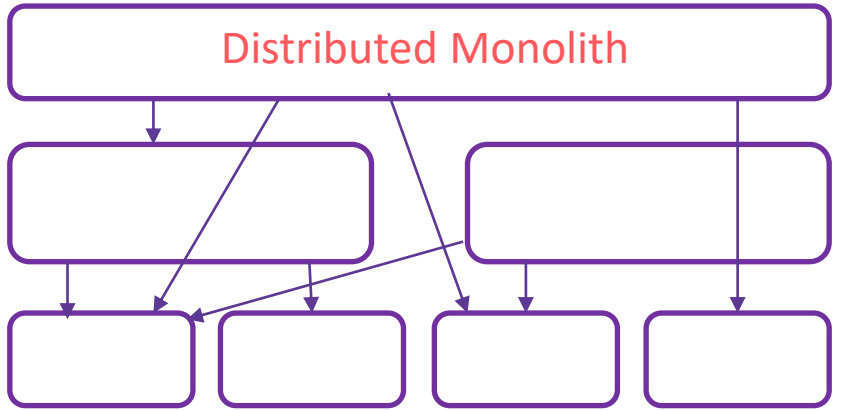
You are working with your team who has been tasked with “stabilizing” a legacy web application that is core to both the operation of the company as well as revenue. Your team is told that it is a priority. When you ask why it is a priority at this time, it is explained to you that there are some defects that are impacting the business’s ability to operate and “workarounds” that once were enough to maintain the business process, no longer are. Someone from IT leadership says, “We can’t add new features because we end up breaking things even though we think we have tested the application end-to-end. Our time to market is abysmal. The business units are getting fed up.” **The rose is now an onion!**

# The Monolith




terazo
Building Platforms That Scale

# Why a microservice architecture?

Some goodness of microservices are:

- Smaller isolated pieces allow for simplified deployment model.
- With loosely coupled design and less dependencies comes simplicity in understanding.
- Share small services that implement cross cutting (shared) functionality. Reusability.
- Easier to test since smaller test footprint.
- Faster ability to isolate and mitigate defects.
- Easier to monitor and manage a microservice.
- Compliments containerization well which allows for scalability, especially in cloud.

# Turning point

When faced with a stabilization or modernization effort, or when faced with the reality that you need a stabilization or modernization effort, embrace microservices to both positively impact the business you work in or the clients you work for.

To be successful in modernizing a legacy application with microservices, we need to embrace the fact that it will involve making a fundamental shift in the way we think, in the processes we follow, in the culture we embrace, in the way we work and sometimes even the people we work with.

# Best Practices

# Decompose using iterative approach

Not sure how to start?

- Meet with the folks who use the legacy application and get a list of all the pain points they experience.
- Work on prioritizing the top five to ten items.
- Take a deep dive into the legacy code and study the code that has the responsibility for implementing the functionality in question.
- Try to understand the intent of the code as written.
- If the intent is unclear (that might be part of the root problem), it's time to meet with the end users/product owner to establish, in clear requirements, what needs to happen.

# Decompose using iterative approach

Not sure how to start?

- Determining where the integration of the microservice should occur.
- Translate your findings to each microservice, building your needed functionality within your microservice.
- Finally, define an integration strategy.
- Follow this pattern until the identified priority pain points have been mitigated by microservice integration. Meet and assign the next set of pain points to be addressed and start the process all over again.

# Decompose using iterative approach

There are other avenues to use when ascertaining where to introduce microservices.

As a development team, review the code together, looking for any:

- “Code smells” that may indicate areas of functionality that may benefit from refactoring to a microservice.
- High traffic/high risk areas of code with no testing.
- External calls to web services, SOAP-based or RESTful, and consider implementing a microservice wrapper for that. This will allow any changes the external service makes to be handled by a microservice and allow great testing as well as individual deployments.

# Get a Product Owner

What is a product owner and why do we need one?

A product owner is one of the keys to success when moving to a microservice architecture. A product owner performs several key duties:

- Providing product vision
- Owning and prioritizing the product backlog
- Involving customers, users, and other stakeholders
- Collaborating with others on the team

The role of product owner is critical for ensuring that the rest of the business and the IT team work together effectively. It also requires significant effort on a daily basis. The product owner provides vision, mentors the team, answers questions, makes decisions about the product, communicates with the broader organization, negotiates resource contentions, coordinates business interaction and serves as a liaison to leaders.

# Get a Product Owner

The product owner is ideally in between the business and the IT teams.

The product owner must be able to communicate different messages to different people about the project at any given time.

The vision the product owner provides is based on:

- Hard business requirements – “we have to send an email to the customer.”
- Implied business requirements – “we hope to double the transaction load in 2 months.”
- Best practices – “Let’s build in a heartbeat check on the microservice.”
- Best practices – “Make sure you write a test for every microservice we build and deploy.”
- Upcoming initiatives – “The Operations team is looking to move forward with containers Q2.”

Becomes the single voice for the product based on what they know.

# Containerize

Containerization reduces wasted resources because each container only holds the application and related binaries or libraries. By allowing more containers in the environment without the need for more servers, containerization increases scalability anywhere from 10 to 100 times that of traditional VM environments.

Containers can be deployed as part of a CI/CD pipeline.

Key benefits:

- Increased portability.
- Improved scalability.
- Simple and fast deployment.
- Increased productivity
- Better Security

# Containerize

Containerization is the natural successor to virtualization.

Portability and consistency are the main drivers.

Orchestration makes all the difference, a greater number of moving parts increases the potential for greater friction.

Containers are perfect for a microservice architecture – like peanut butter and jelly!

Microservices use containers to deliver smaller, single-function modules, which work to create scalable applications. With this approach, there is no need to build and deploy an entirely new software version every time you change or scale a specific function.

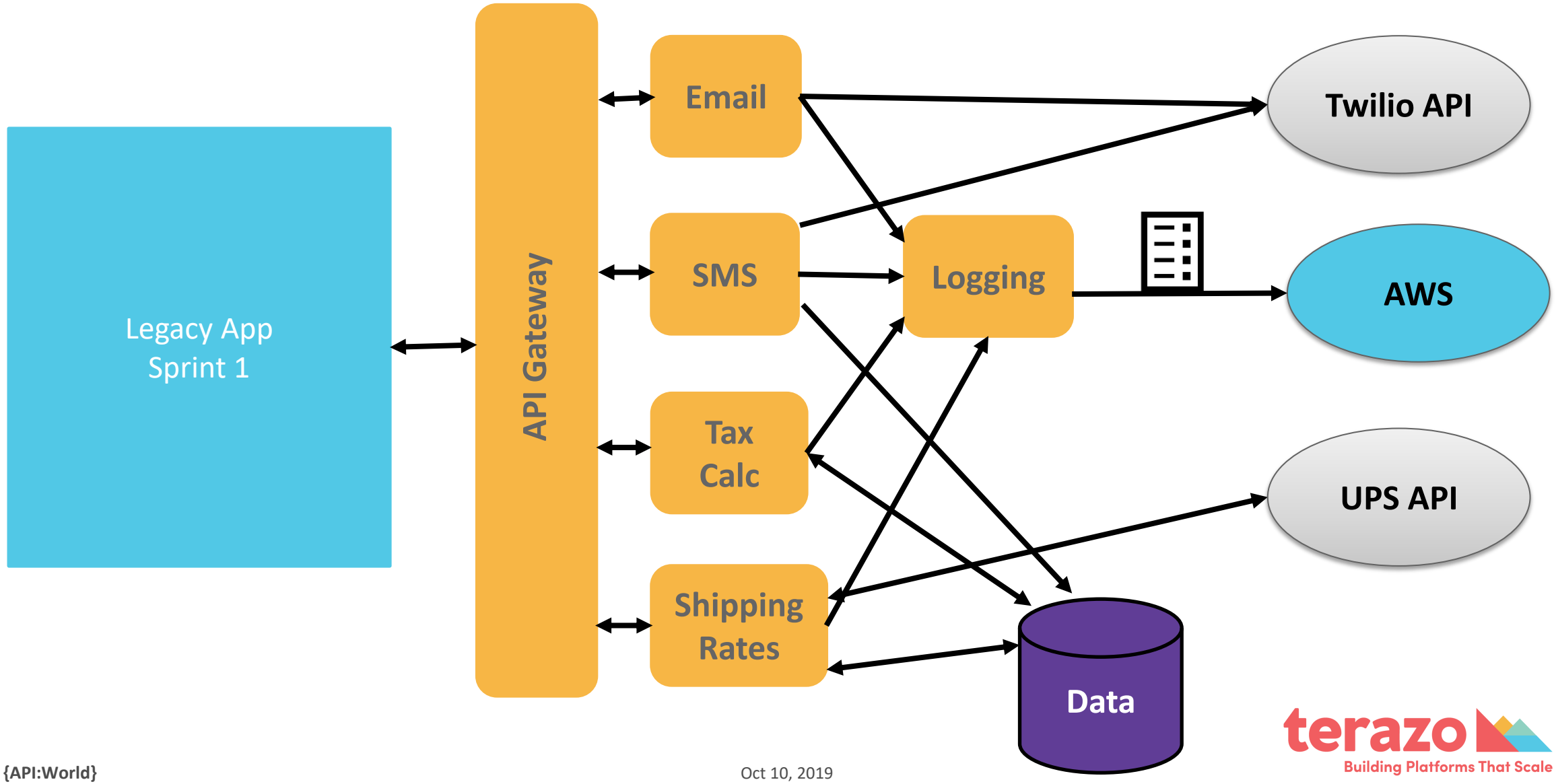
# ★ Rough design up front

Big Design Up Front (BDUF) is a tenant of the Waterfall methodology of application development.

Rough Design Up Front is different in strategy and intent.

Significant parts of a solution can, and should, be designed up front. Not every project is so complex and uncertain that it must be evolved from scratch — In fact most are not. Experienced Product Owners and Technical Architects/Leads can usually create a rough design for up to 75% of a solution in the first few iterations of a project.

# Rough design up front



# Follow good API patterns

Some of the common mistakes when building APIs for REST microservices include:

- Designing the API based on the client
- Designing the API the way you would design a class – what I call the MOD (Many Overloads Design)
- Testing the microservice by testing the legacy application

Here are some good patterns and practices to follow when designing and developing APIs:

- Use plurals - plurals help avoid confusion when we are talking about getting single resource or a collection. => */cars/car*
- Use nouns not verbs => /customer vs /getAllCustomers

# Follow good API patterns

Good patterns and practices to follow when designing and developing APIs con't:

- Use plurals - plurals help avoid confusion when we are talking about getting single resource or a collection. => */cars/car*
- Use the right HTTP Methods => GET, POST, PUT/PATCH and DELETE
- Use parameters – Sometimes we need and API to return data by more than id, we should make use of query parameters to design the API.
  => products?name='Wild Widget' should be preferred over /getProductsByName
- Use proper HTTP codes - Most of us only end up using two—200 and 500! This is certainly not good. => 200 OK, 201 CREATED, 202, ACCEPTED,400 BAD REQUEST etc.
- Versioning - Versioning of APIs is very important. User header versioning or URL.

# Follow good API patterns

Good patterns and practices to follow when designing and developing APIs con't:

- Use pagination – exposing a huge amount of data over a REST API microservice can slow down the consumer of your service and impact user experience. = /products?limit=25&offset=50
- Proper error messages
- The GET method should not be used to alter state – Instead use the PUT, POST and DELETE methods. GET /orders/737272/process

# Iterative development

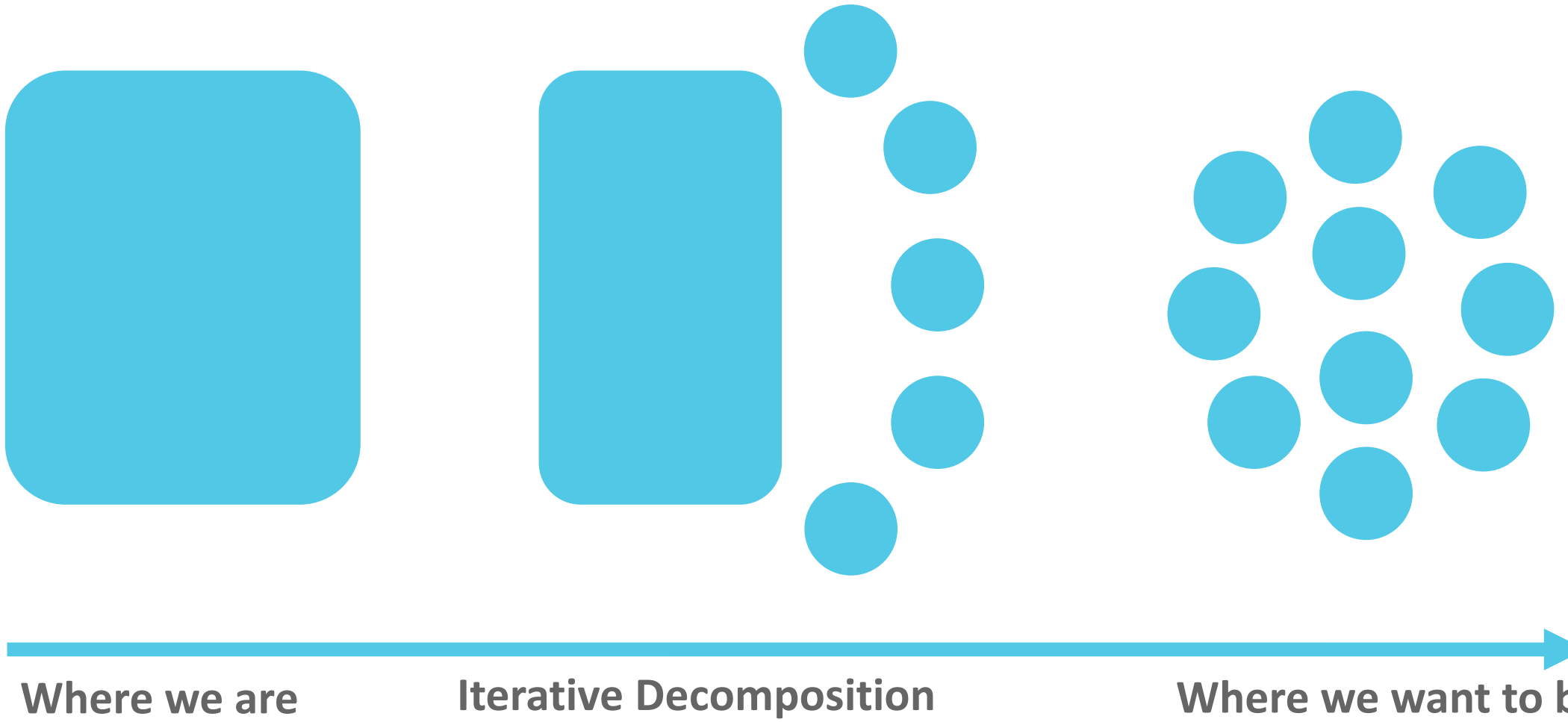
Iterative development is now becoming common practice because it better fits the natural path of progression in software development. Instead of investing a lot of time/effort chasing the 'perfect design' based on assumptions, the iterative approach is all about creating something that's 'good enough' to start and evolving it to fit the user's needs.

At the end of the iteration, working code is expected that can be demonstrated for a customer.

Example:

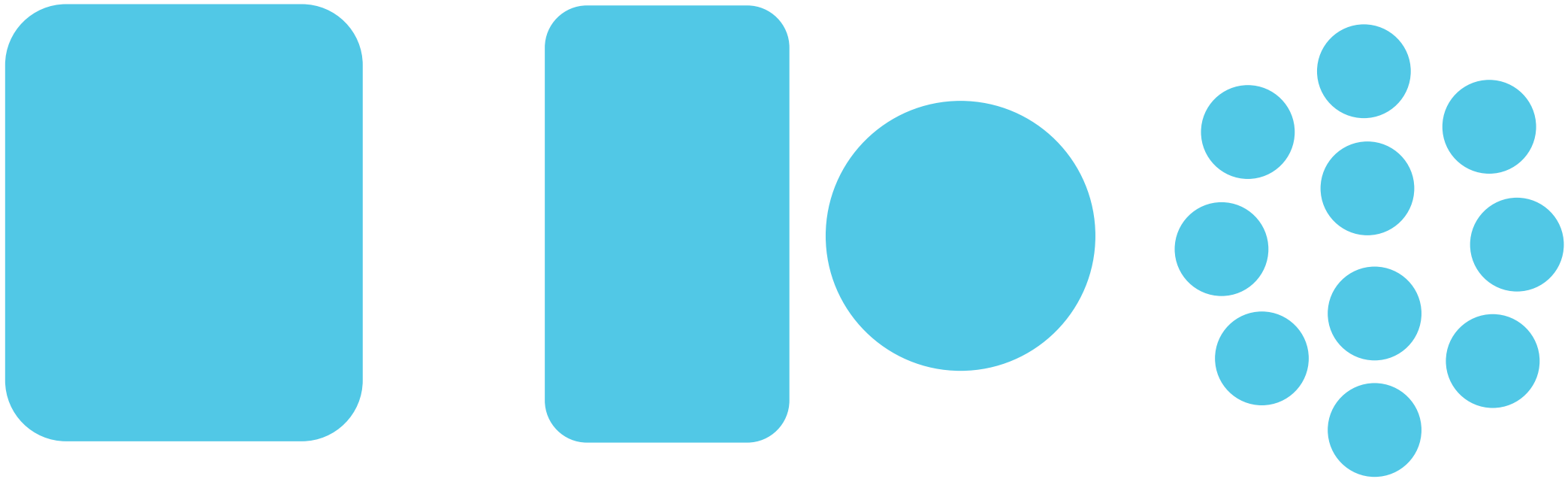
Review/add user stories -> Update rough design -> write tests -> develop -> CI -> CD -> Test

# Create a road map and decompose with it in mind




terazo
Building Platforms That Scale

# Create a road map and decompose with it in mind



**Where we are**

**Iterative decomposition to monolith**

**Where we want to be**

terazo
Building Platforms That Scale

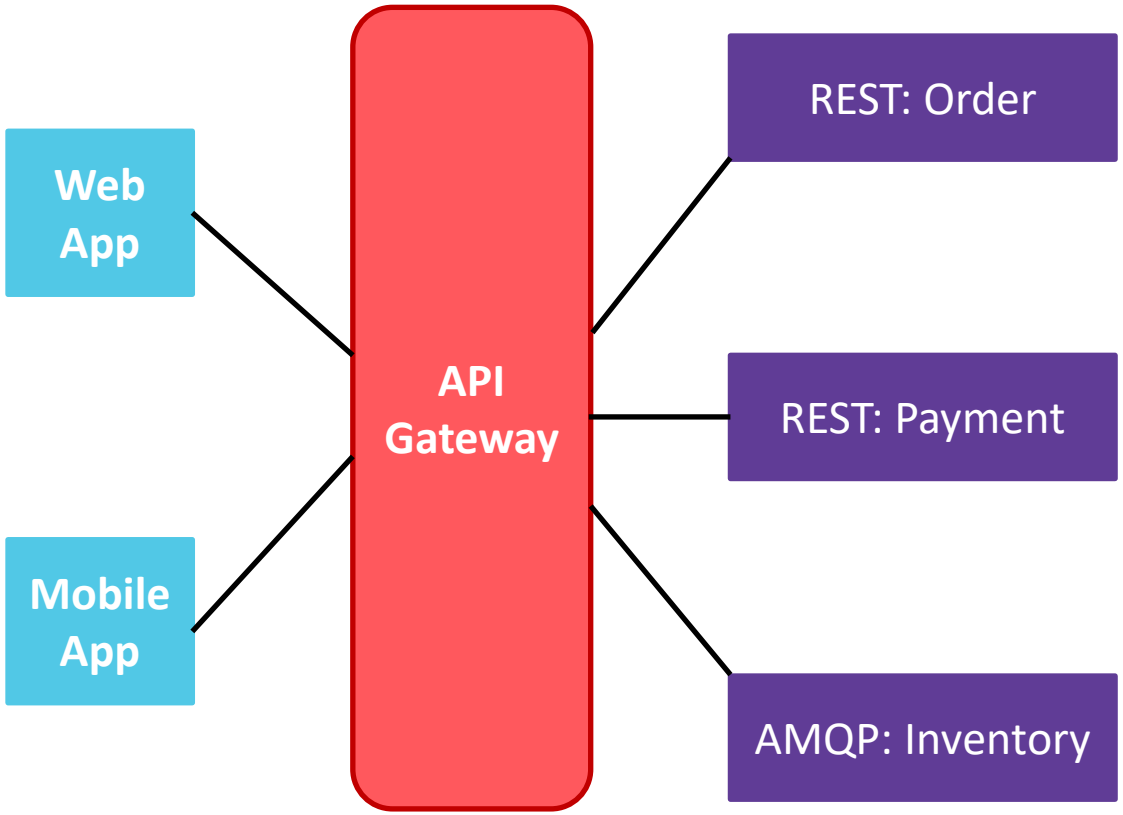
# Implement an API Gateway

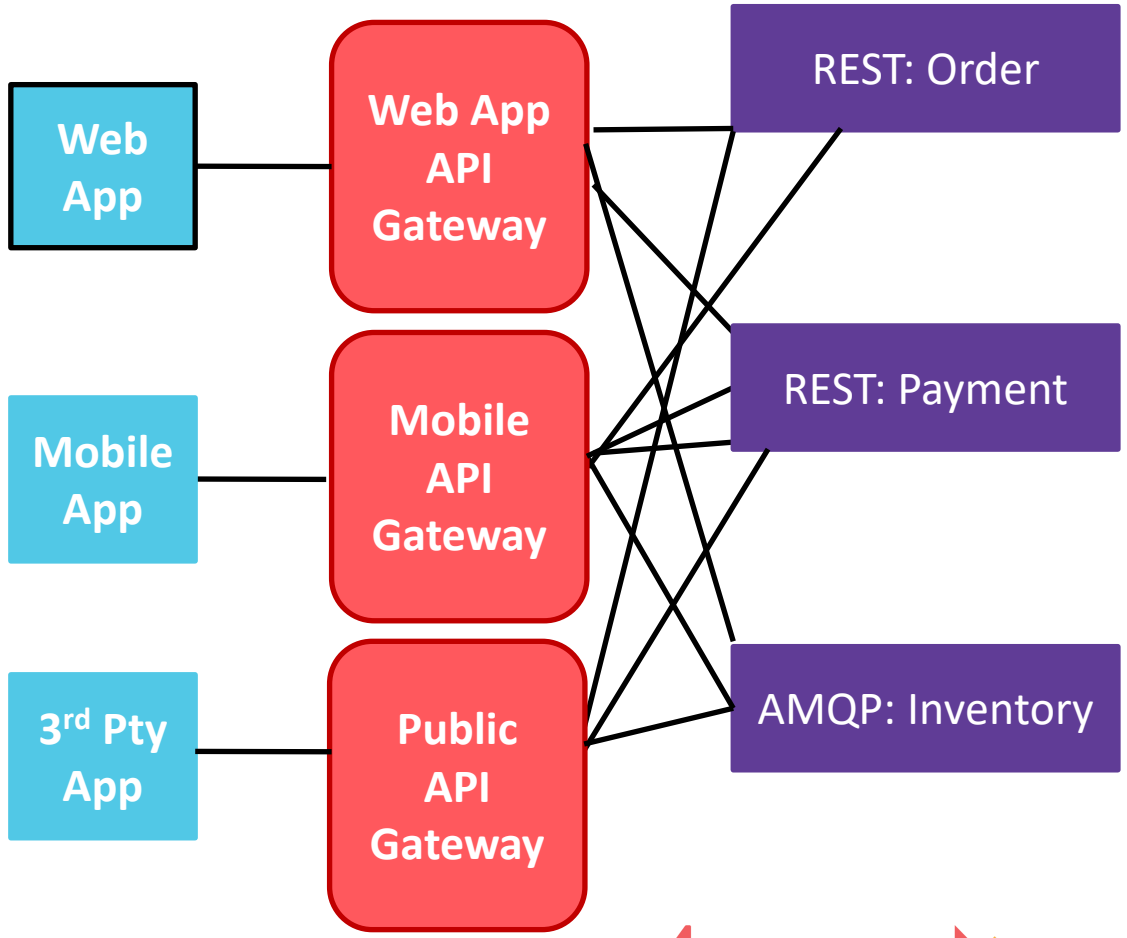
An API gateway is technology that sits in front of an API and acts as a single point of entry for a defined group of microservices. It handles routing,

Some benefits:

- Insulates the clients from how the application is partitioned into microservices.
- Insulates the clients from the problem of determining the locations of service instances.
- Provides the optimal API for each client.
- Reduces chattiness. The API gateway enables clients to retrieve data from multiple services with a single round-trip. Fewer requests also means less overhead and improves the user experience. An API gateway is important for mobile applications.
- Simplifies the client by moving logic for calling multiple services from the client to API gateway.
- Translates from a “standard” public web-friendly API protocol to whatever protocols are used internally.

# Implement an API Gateway



# Implement an API Gateway

Warning!!

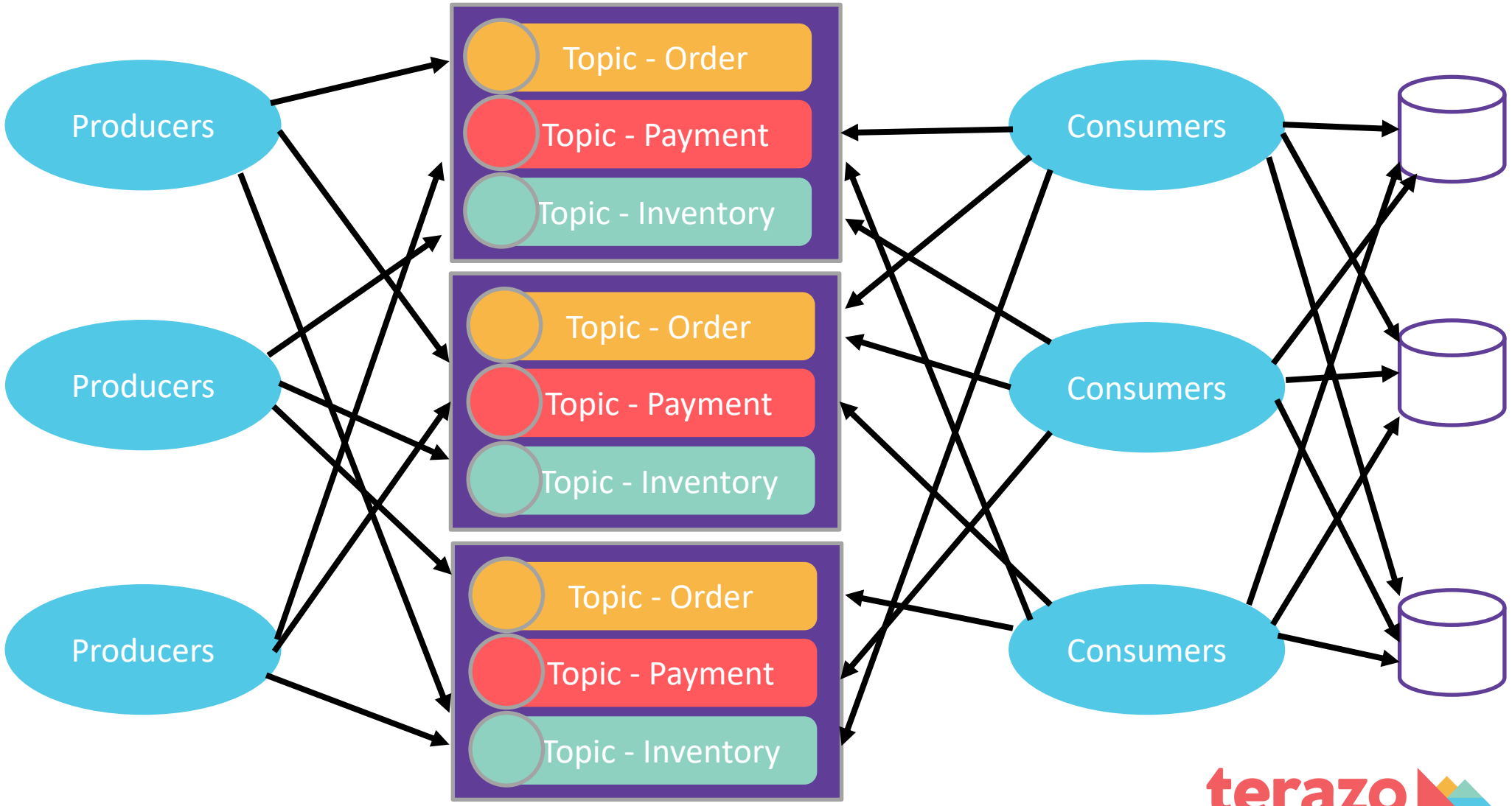
If not implemented correctly, the API gateway can become as costly to maintain as the legacy application.

It is yet another moving part that must be developed, deployed and managed.

# Work towards an event driven architecture

- Since microservice architecture is an approach to developing an application as a suite of small independently deployable services built around specific business capabilities, it plays nicely with event driven architectures and event streams.
- When moving from legacy applications to a microservices architecture, a common architecture pattern is event sourcing using an “append only” the event stream. With event streaming tools, events are grouped into logical collections of events called Topics. Topics are partitioned for parallel processing. You can think of a partitioned Topic like a queue, events are delivered in the order they are received.

# Work towards an event driven architecture



# Write good tests

Testing helps coach good development. Even if you have a simple logging service that receives a message and logs it, it should send a status. Alternatively  you could implement an API that reads the log file based on the log you just created. The point is write tests for your microservices.

1. Unit Tests/Functional Testing
2. Performance Testing

- Keep the testing code compact and readable
- Cover as much of the range as possible to show positive cases and especially erroneous code paths
- Don't mock a type you don't own!

# Summary

Decompose using iterative process.

Get a product owner.

Containerize.

Rough design up front

Follow good API development patterns.

Iterative development.

Create a roadmap and decompose with it in mind.

Implement an API gateway.

Work towards an event driven architecture.

Write good tests.

# Questions


**Contact Information**

**Kevin Israel**

**Market Technical Director**

kevin.israel@terazo.com

https://www.linkedin.com/in/kevinisrael/

www.terazo.com


terazo

Building Platforms That Scale