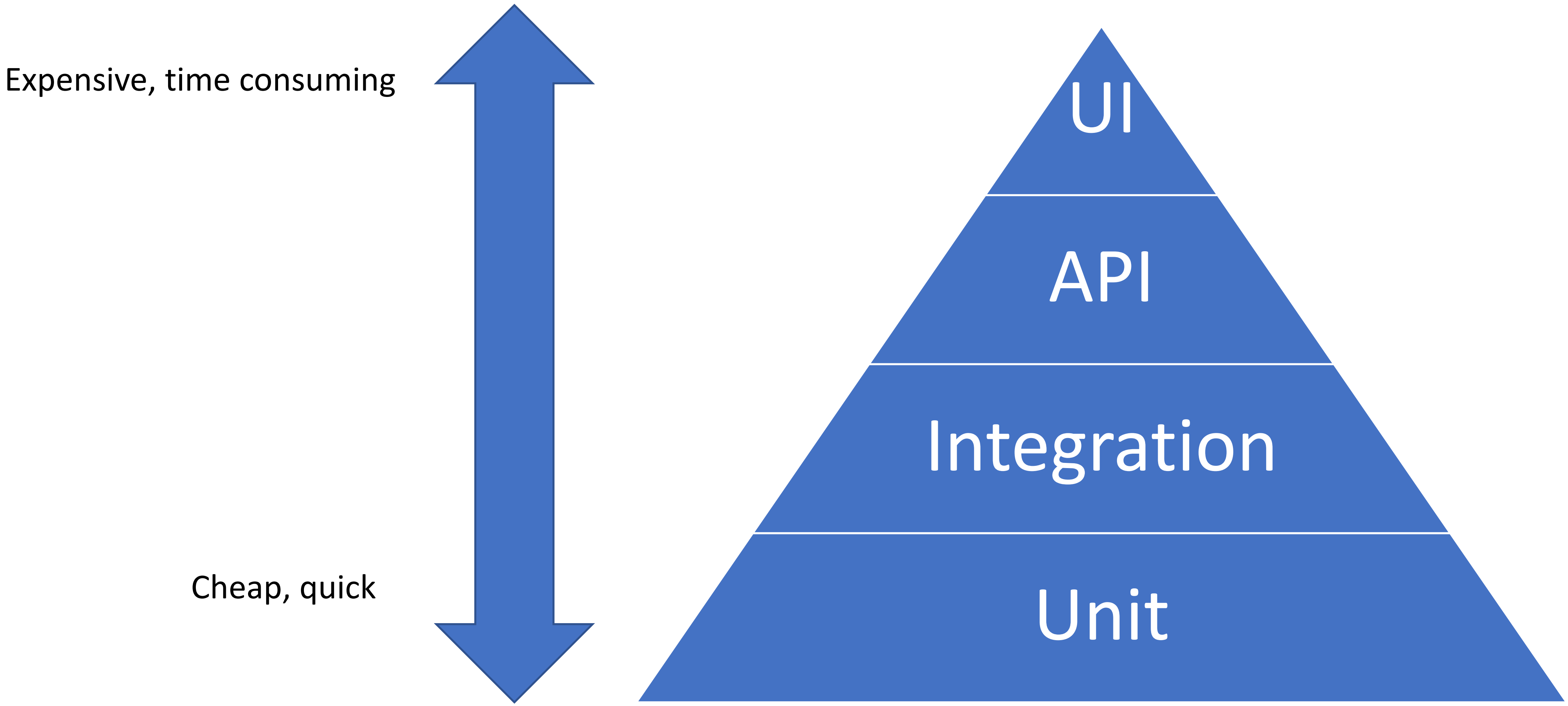
Testing Microservices

Some opinionated and possibly wrong suggestions

The Classic Testing Pyramid



Will this work for Microservices?

Remember that you use Microservices to build cloud services.

You aren’t shipping code to clients

You have a complicated live system that is always running

Invert the Pyramid

Embed testing in your design at each level

Inverted Pyramid: UI and API Testing

Develop user stories up front

Create mockups for the user interface

Develop your APIs based on the user stories.

Implementing the mockup creates your API reference implementation

Implement UI and API tests from the beginning

Use off the shelf tools like Selenium

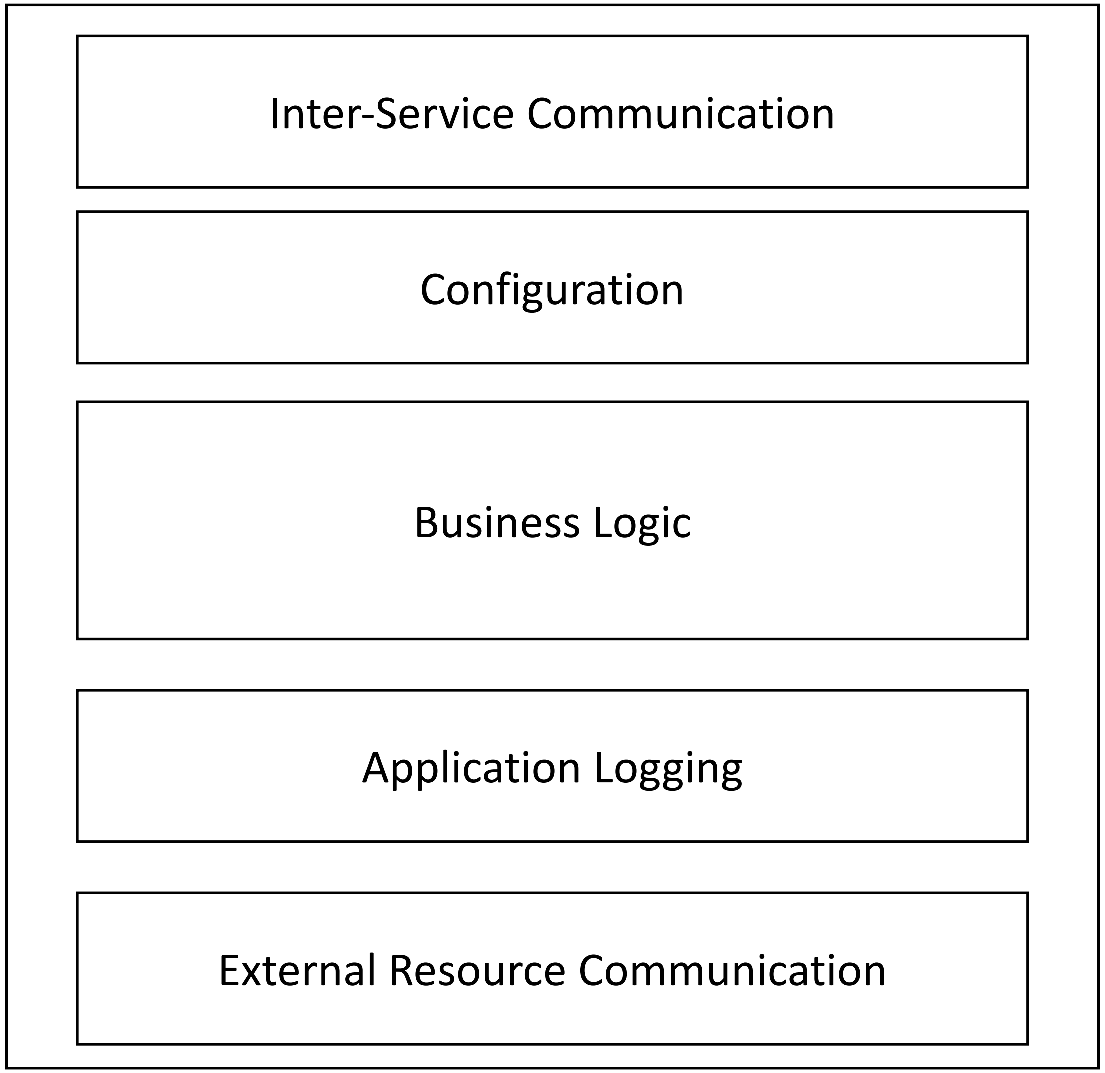
Inverted Pyramid: Integration Testing

- Decide on your messaging and coordination strategy
- Enumerate your microservices
- Create internal APIs and data models for each service
- Create skeletons for each microservice
- Design API (“contract”) tests for your system

Inverted Pyramid: Unit Testing

- Internally, microservices follow patterns
- Use these patterns to create (object oriented) abstractions
- Extend your abstractions for each service instance
- Test the whole service, not just the specific business logic

Anatomy of a Microservice



- **Inter-Service Communication** connects the microservice to other microservices via the control plane and data (messaging) plane.
  - REST, gRPC SDK, RabbitMQ, Kafka, ...
  - Security
  - Instrumentation communications
- **Configuration** contains the service’s operational parameters.
- **Business Logic** implements what the service actually does.
- **Application Logging** logs the service’s operations for monitoring and debugging.
- **External Resource Communication** connects the microservice to its DB or an external data store.

A Bold Hypothesis

- Most important bugs don’t come from your own bad business logic.

The Origin of Bugs

- Bugs come from working code that fails to meet non-functional requirements like performance or security.
- Bugs come from the parts of your service that you didn’t write: SDKs, other generated code, libraries, and other dependencies
- Or they come from the environment where your services run
- Or they come from other unexpected events and behaviors that are hard to reproduce

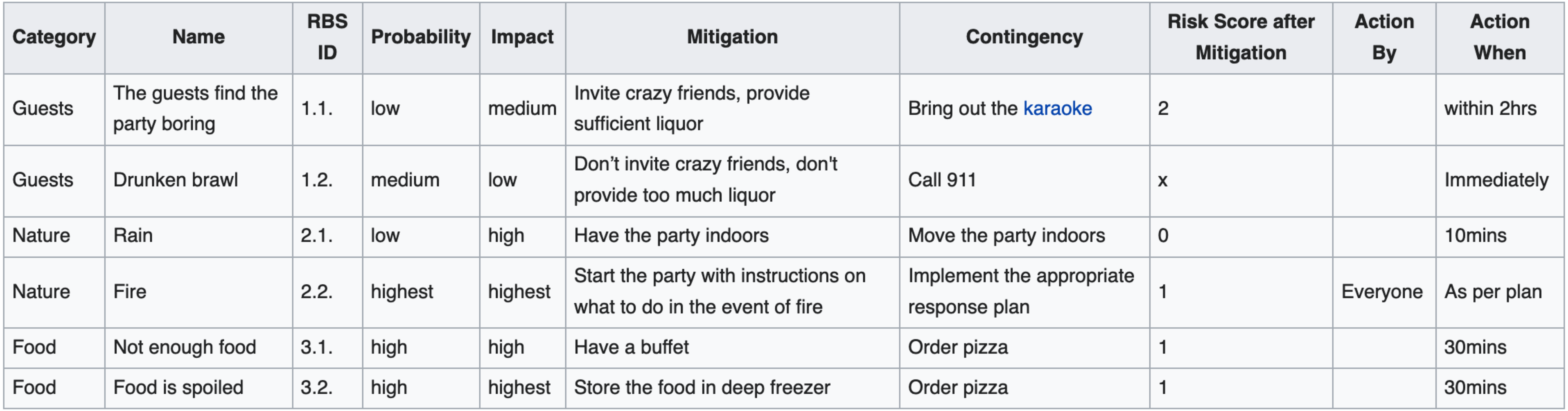
Therefore, you need to develop tests around a global view of risks to your system

A Risk Register Approach to Testing

Risk registers are used in project management to enumerate risks (ISO 31000, for instance)

Each risk has a description, severity level, probability, mitigation, and contingency

Example of a Risk Register



Category	Name	RBS ID	Probability	Impact	Mitigation	Contingency	Risk Score after Mitigation	Action By	Action When
Guests	The guests find the party boring	1.1.	low	medium	Invite crazy friends, provide sufficient liquor	Bring out the karaoke	2		within 2hrs
Guests	Drunken brawl	1.2.	medium	low	Don’t invite crazy friends, don't provide too much liquor	Call 911	x		Immediately
Nature	Rain	2.1.	low	high	Have the party indoors	Move the party indoors	0		10mins
Nature	Fire	2.2.	highest	highest	Start the party with instructions on what to do in the event of fire	Implement the appropriate response plan	1	Everyone	As per plan
Food	Not enough food	3.1.	high	high	Have a buffet	Order pizza	1		30mins
Food	Food is spoiled	3.2.	high	highest	Store the food in deep freezer	Order pizza	1		30mins

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

Risk Register Example for Project Management

- Risk: lead developer leaves project for a different job
- Severity: high
- Probability: medium-high
- Mitigation: groom successors
- Contingency: promote a successor

Apply This Approach to Your System

- Think about failure events at all probability levels
- Develop mitigation and contingency plans
- Map these to tests
- Test failures trigger risks

Example: Customer is charged twice for the same purchase

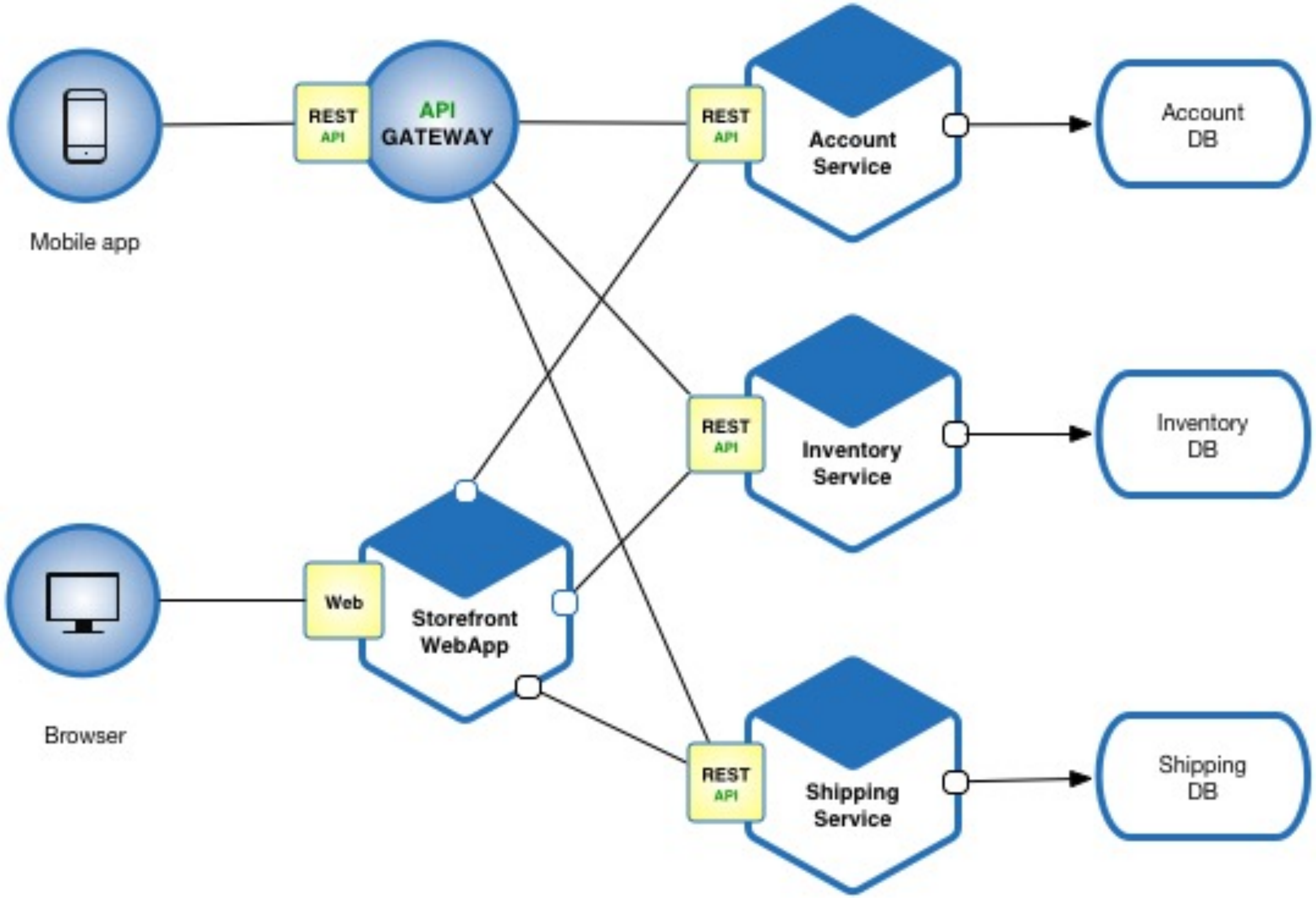
- Probability: low if you follow good design patterns
- Severity: medium if you can undo the doubled payment; otherwise, high
- Mitigation: adopt appropriate patterns, develop tests to detect occurrence
- Contingency: Notify customer, credit card company

Approaches to Microservice Integration and Unit Testing

First, choose your communication and coordination strategies

We’ve called these the Control Plane and the Data Plane in previous lectures

The Basic REST Approach



https://microservices.io/patterns/microservices.html

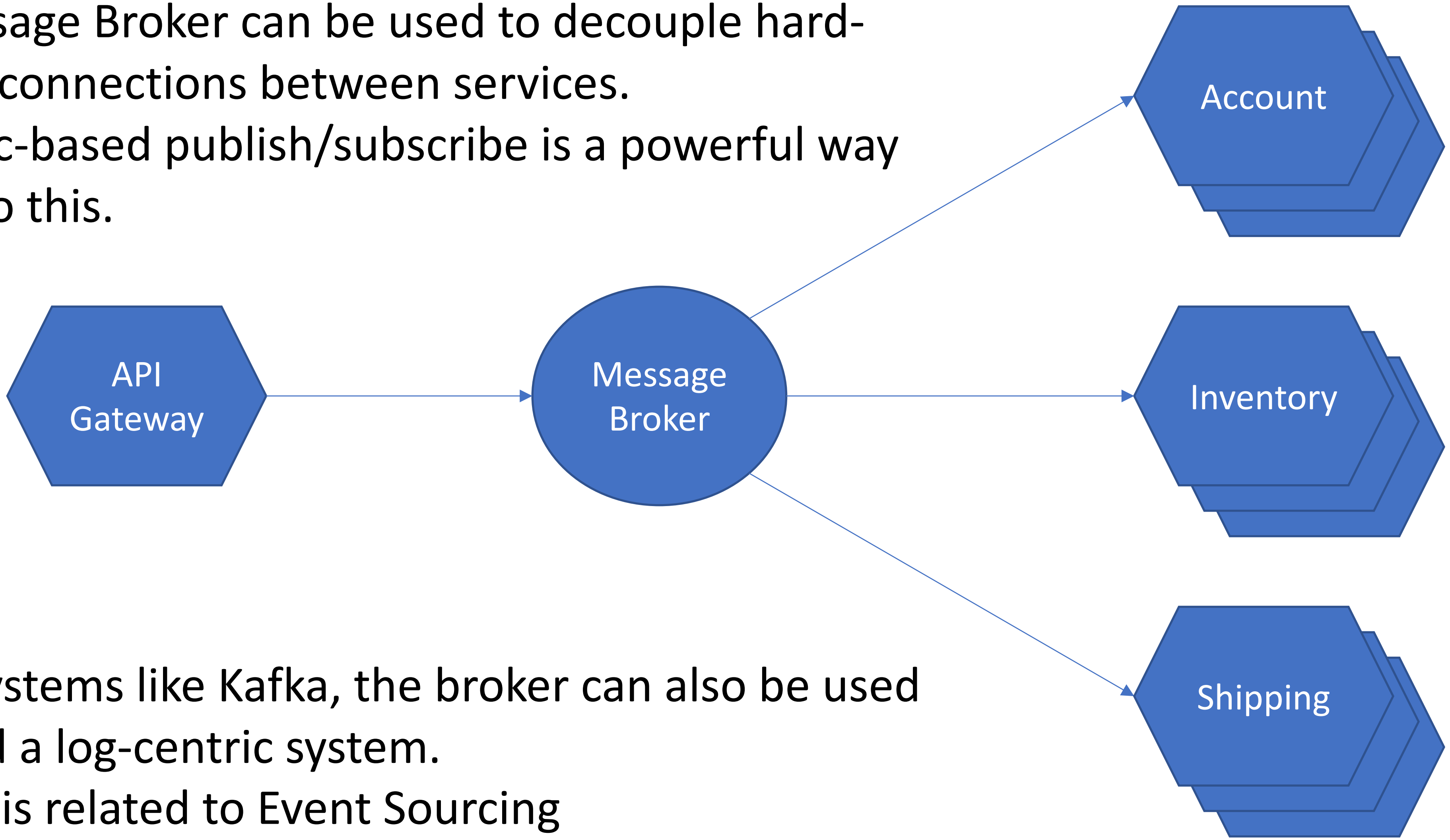
Critique of the Basic REST Approach

- Control logic is embedded in particular services like the storefront
- Service paths are accessed by instance rather than by function or type
- How do you know if a service is down?
- It's static and brittle
- It is hard to test

Messaging and Log-Centric Approaches

A Message Broker can be used to decouple hard-coded connections between services.

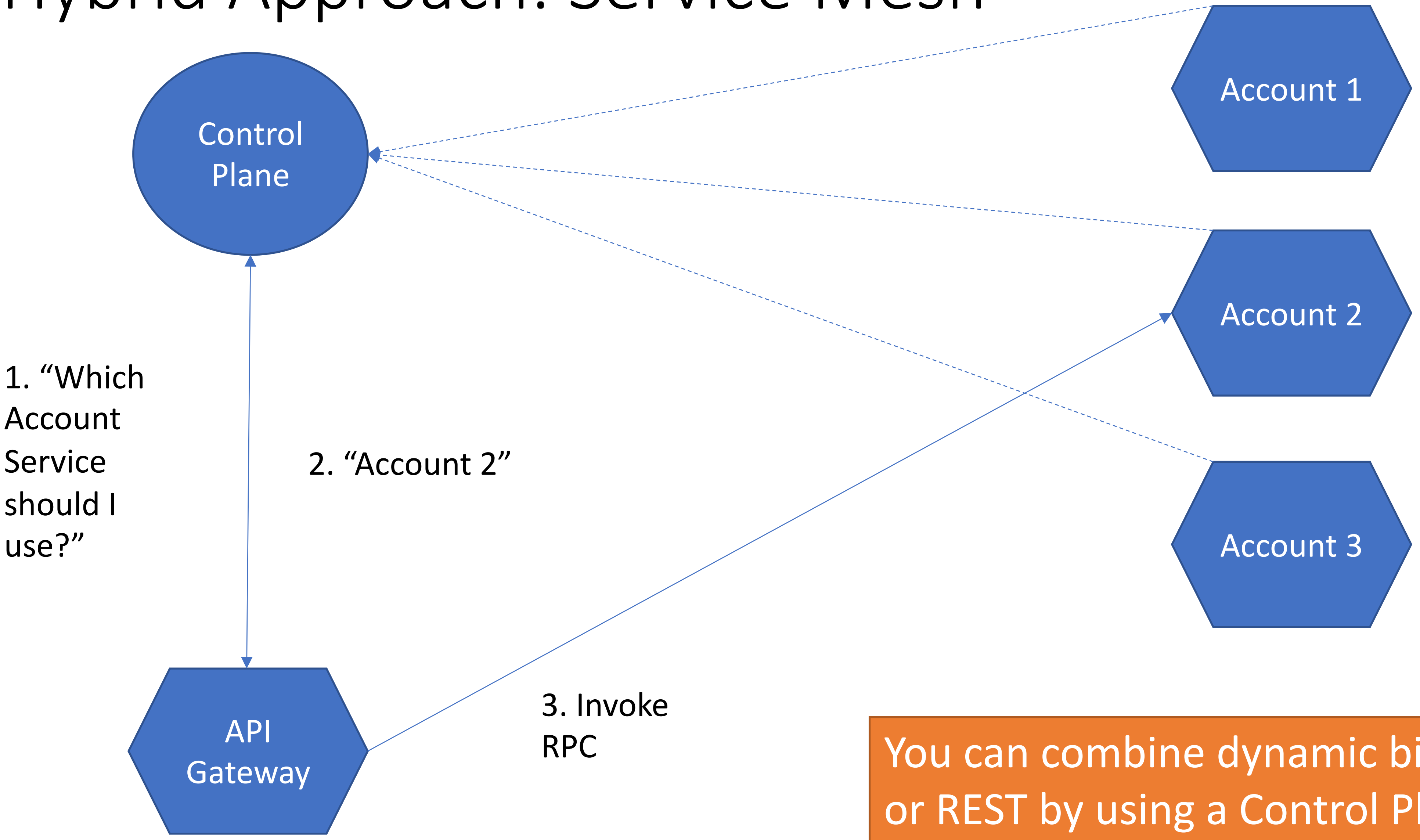
- Topic-based publish/subscribe is a powerful way to do this.



With systems like Kafka, the broker can also be used to build a log-centric system.

- This is related to Event Sourcing

A Hybrid Approach: Service Mesh



You can combine dynamic binding and RPC or REST by using a Control Plane system like Consul.

Foundations of Testing Microservices: Have a Clean Architecture

If using messaging, use it consistently. All communications go through the message bus.

If using a separate Control Plane, all services use the control plane service to coordinate.

Avoid having some services use one communication mechanism and other services use another

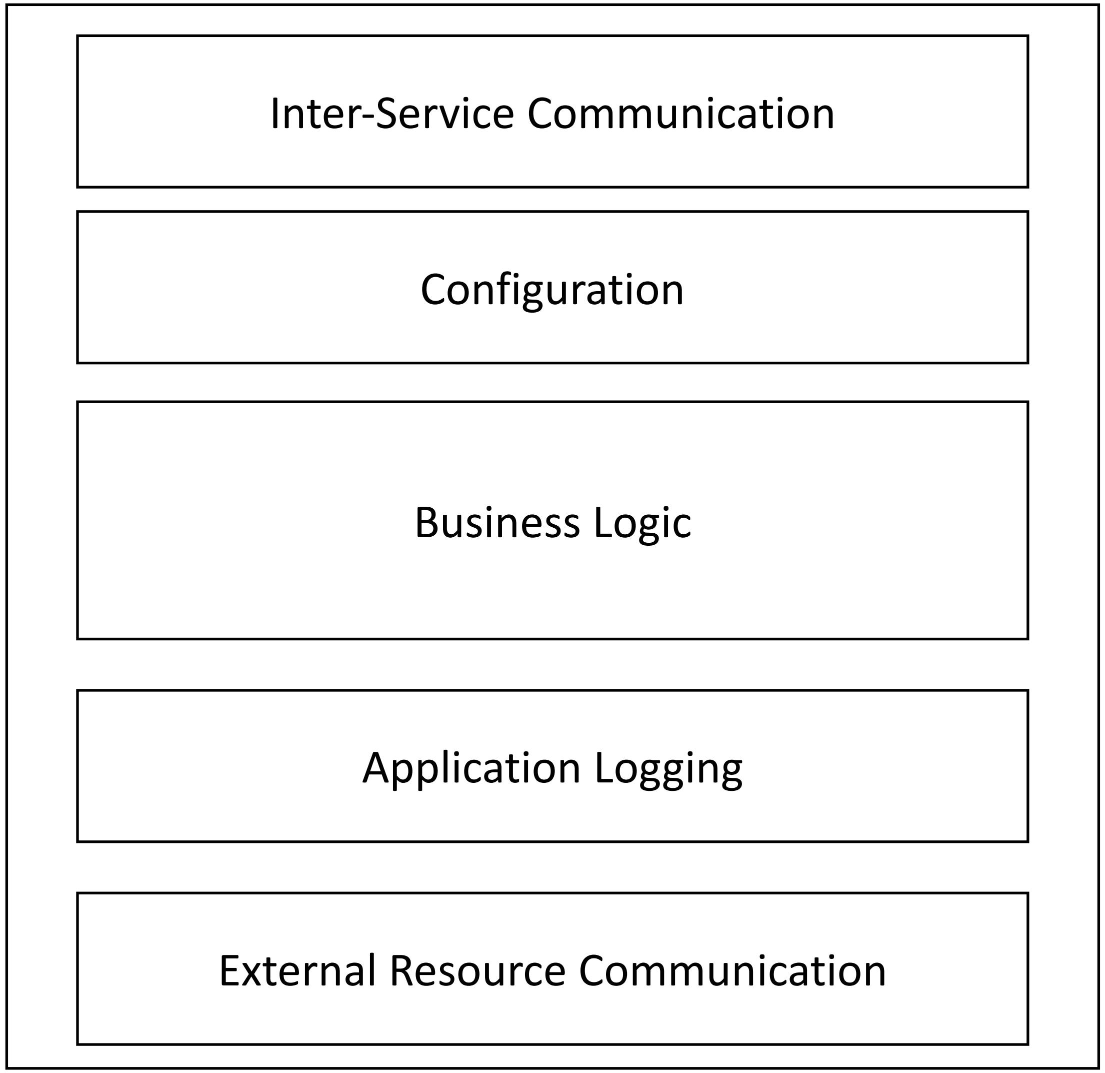
Foundations of Testing Microservices: Topic-Based Publish/Subscribe Is Powerful

- Pub-sub systems can send the same message to multiple recipients.
- You can send real messages to both your production and in-testing components.
- Test components receive real messages and real message loads, so you can see how they behave.
- This is a foundation for Canary integration tests (more in a moment)

Foundations of Testing Microservices: Start with the Interfaces and Messages

- Define your microservices by API and/or by the message format and data model.
- What messages does the service receive?
- What messages does the service return?
- If you have this, you can mock services and test API changes

Anatomy of a Microservice



- **Inter-Service Communication** connects the microservice to other microservices via the control plane and data (messaging) plane.
  - REST, gRPC SDK, RabbitMQ, Kafka, ...
  - Security
  - Instrumentation communications
- **Configuration** contains the service's operational parameters.
- **Business Logic** implements what the service actually does.
- **Application Logging** logs the service's operations for monitoring and debugging.
- **External Resource Communication** connects the microservice to its DB or an external data store.

Development Strategy for Effective Unit and Early Integration Testing

Write the business logic layer last.

You may be able to create the other layers by extending a common set of base classes.

✓ Use unit tests at all layers

Unit Testing Strategy

- Don’t limit unit tests to the business logic layer
- Test what happens when you get failures in other layers
- Log problems so that they can be mapped to risks
- Use log aggregation to get a global view of the system

Integration and Deployment Testing

Deployment Pattern #1: Big Bang

- Take the old system down
- Deploy the new system (or subsystem) all at once
- Apologize for downtime
- If you have enough (cloud) resources and a reproducible deployment, there really is no need to do this (see next slide)

Deployment Pattern #2: Blue-Green

Two versions of your system or subsystem are running separately

Route traffic from old (blue) to new (green) system or subsystem

Roll back to blue if you encounter problems

Deployment Pattern #3: Canary

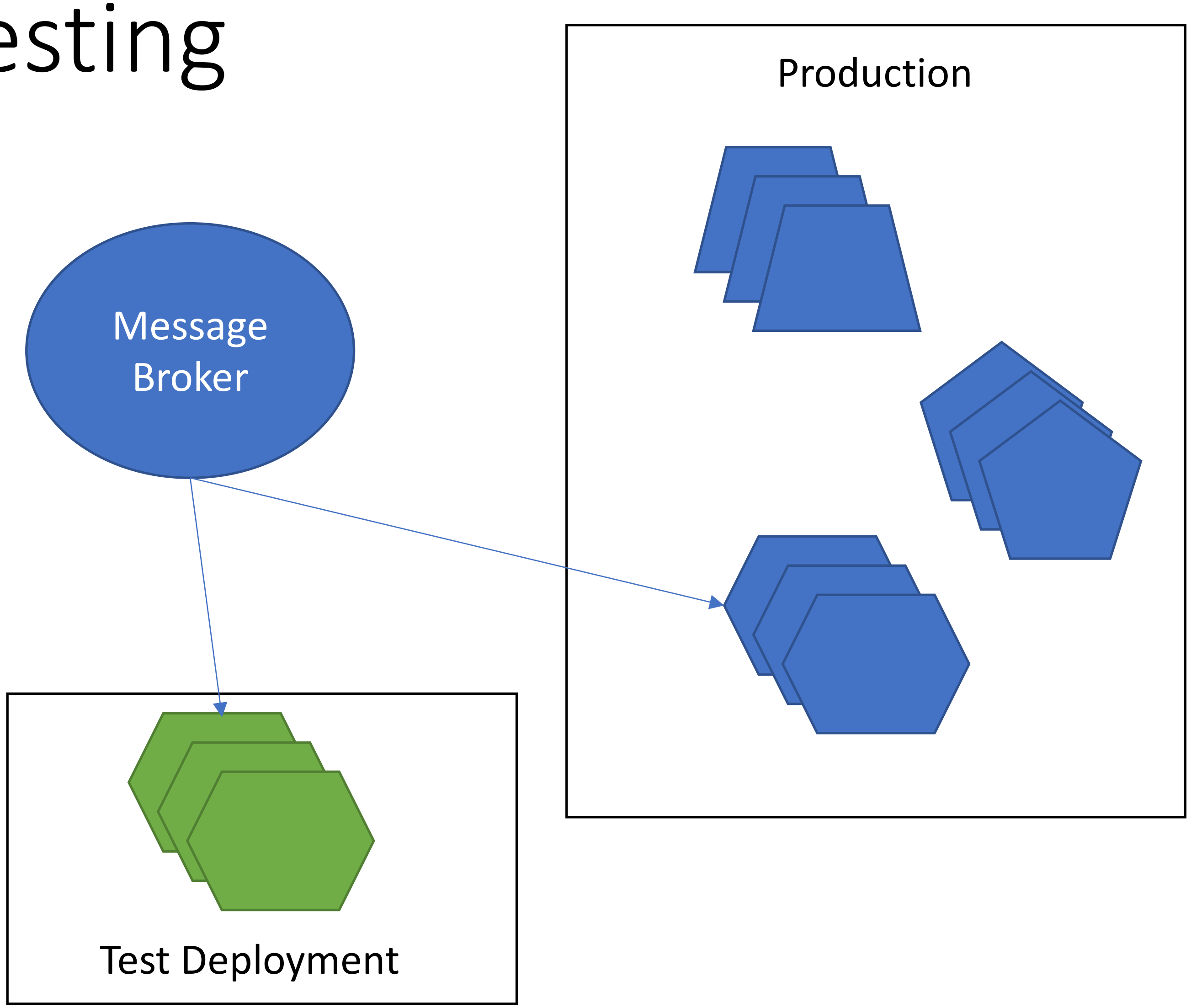
Like blue-green, you maintain two (sub)systems, old and new

Gradually send some traffic to the new deployment.

If everything works, increase traffic to new deployment until you have entirely replaced the old deployment

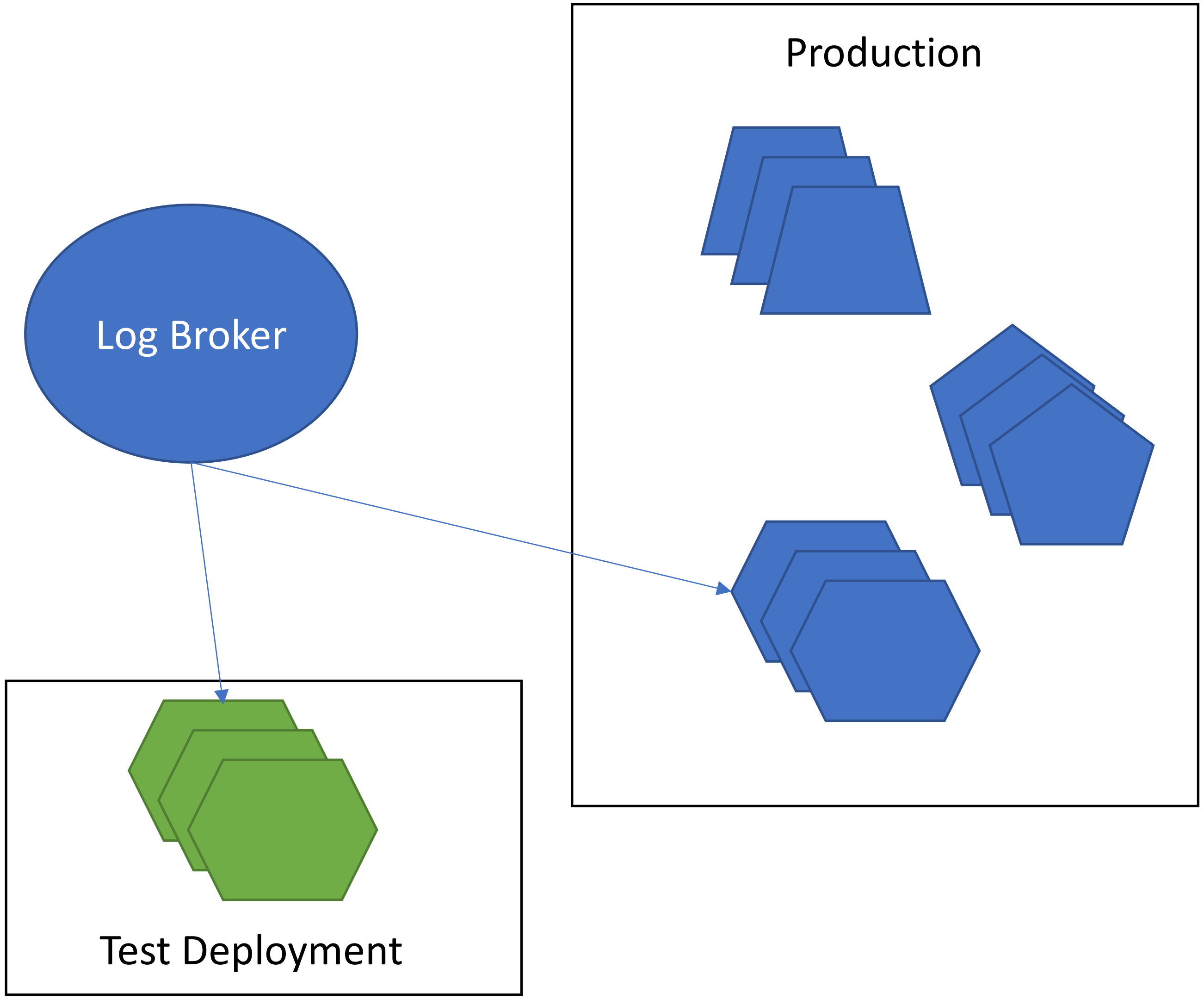
Messaging Systems Enable Blue-Green and Canary Integration Testing

- Pub-sub allows you to send the same message to multiple recipients
- Use this to send to both the production and test systems
- Test your service under real conditions without disrupting production



Log-Centric Systems Enable Replay, Reproducibility

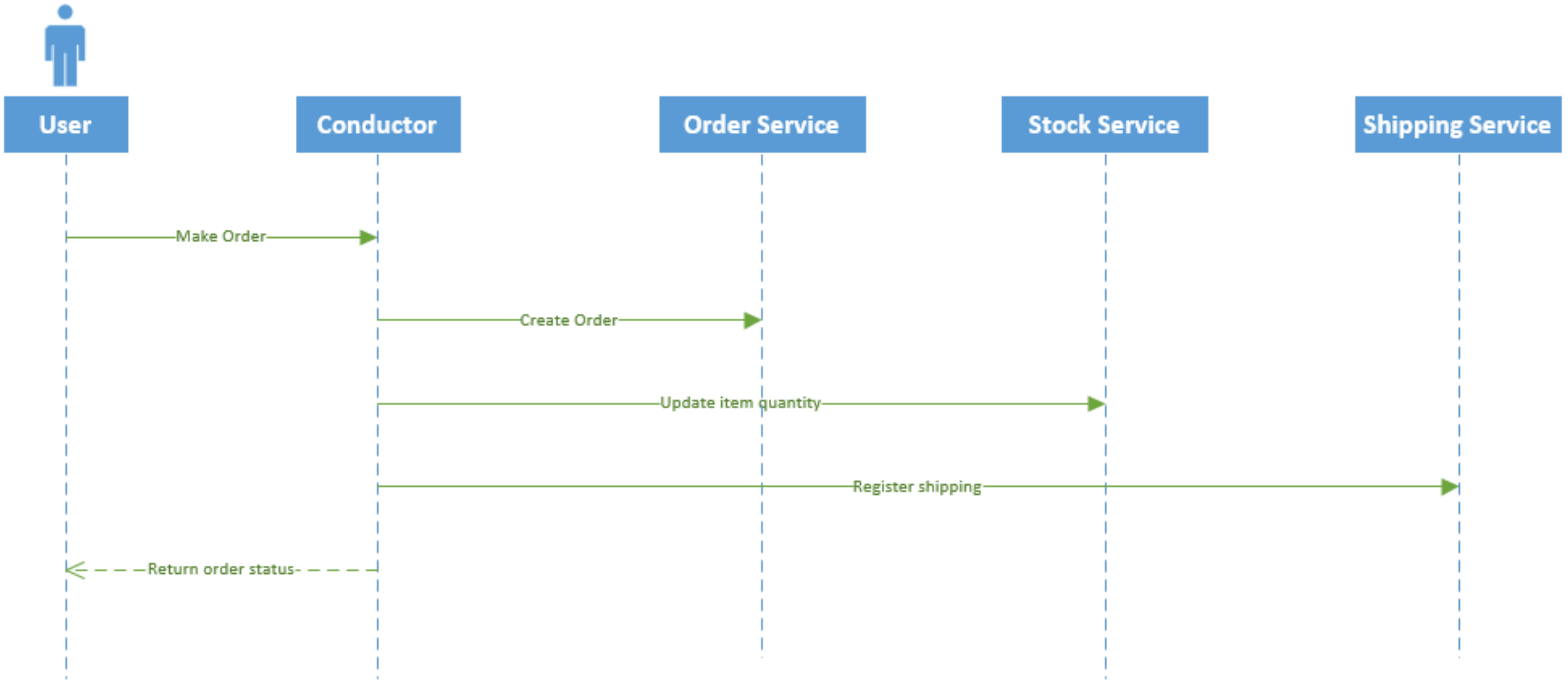
- Test “green” services against historical logs
- Replay logs to help recreate bugs



Testing Subsystems

- A particular interaction like a transaction may span multiple microservices
- Don’t waste time trying to figure out how to implement these yourself
- Instead, use design patterns to help define your implementations consistently
- For testing, deploy all the services associated with the pattern as a subsystem for testing

Distributed Transactions Are a Challenge for "One DB Per Service" Microservices



https://medium.com/swlh/microservices-architecture-what-is-saga-pattern-and-how-important-is-it-55f56cfedd6b

Distributed Transactions

Problem #1: No one database

Problem #2: Transactions can be long-lived, so you don't want to lock resources

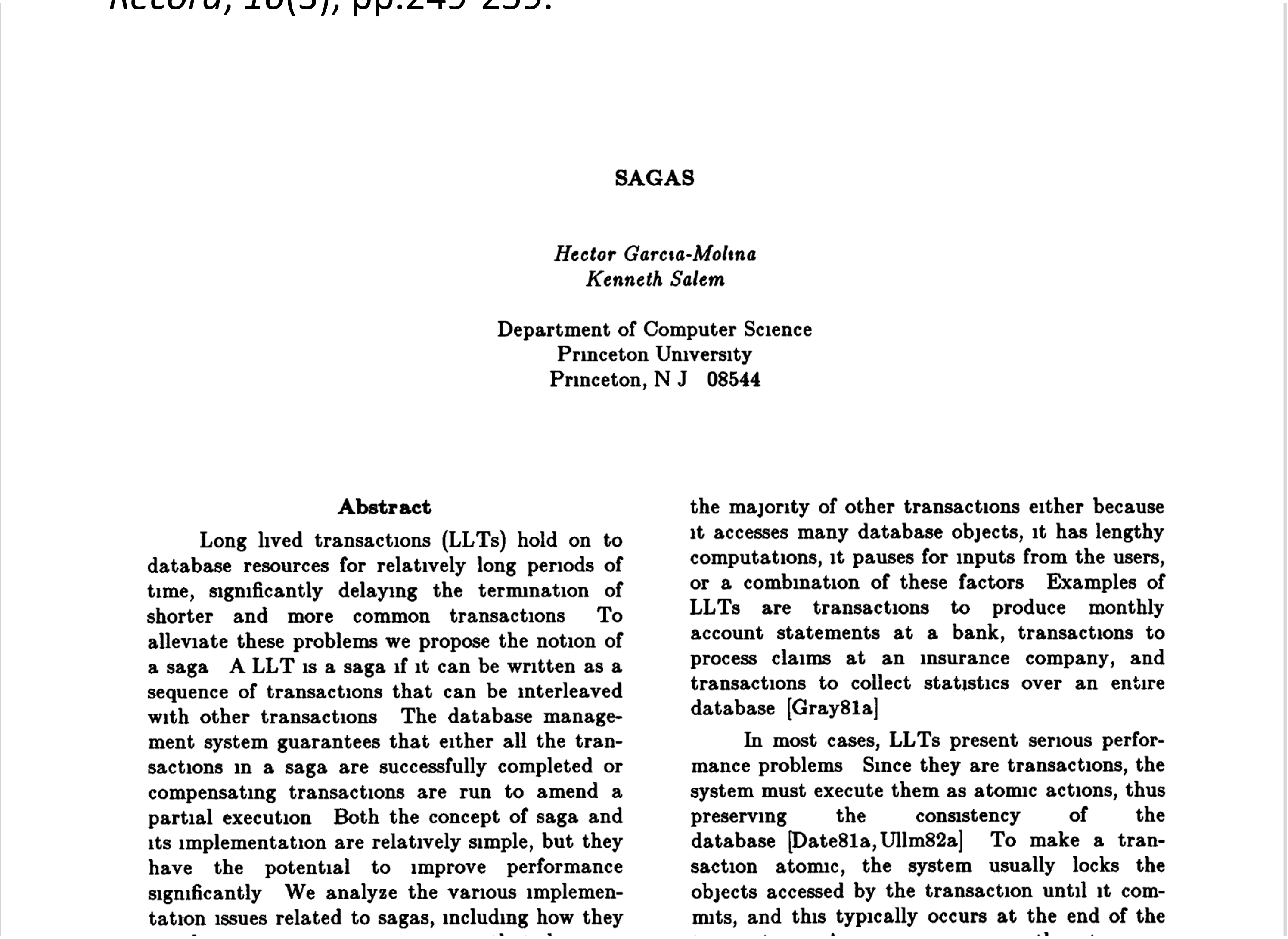
Solution, Part 1: Break the transaction into atomic steps

Solution, Part 2: Have compensatory operations that can undo each operation.

Garcia-Molina, H. and Salem, K., 1987. Sagas. *ACM Sigmod Record*, *16*(3), pp.249-259.

Saga is an example of an academic solution that suddenly became relevant decades later.

Saga: long-lived transaction



SAGAS

Hector Garcia-Molina
Kenneth Salem

Department of Computer Science
Princeton University
Princeton, N J 08544

Abstract

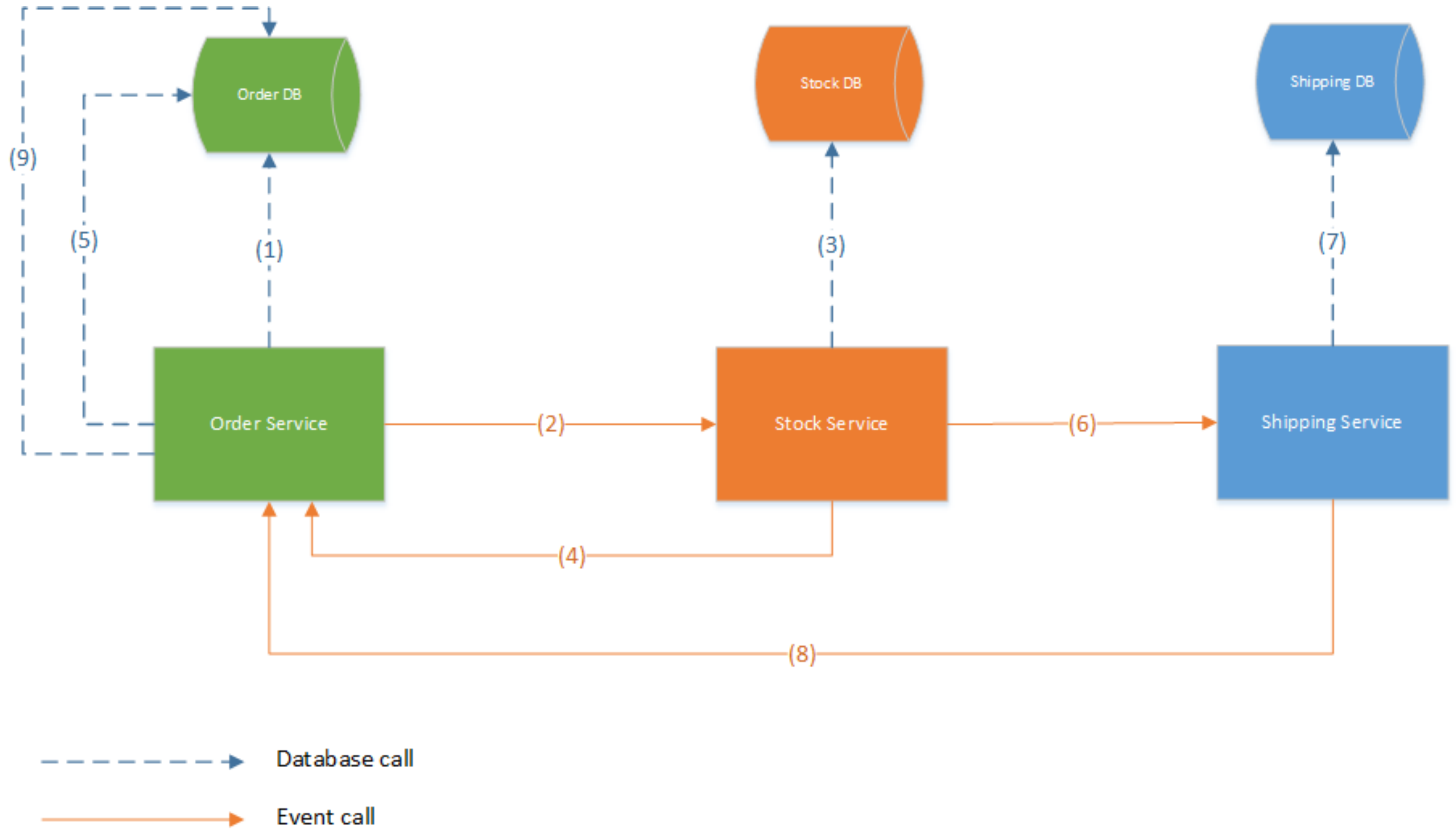
Long lived transactions (LLTs) hold on to database resources for relatively long periods of time, significantly delaying the termination of shorter and more common transactions To alleviate these problems we propose the notion of a saga A LLT is a saga if it can be written as a sequence of transactions that can be interleaved with other transactions The database management system guarantees that either all the transactions in a saga are successfully completed or compensating transactions are run to amend a partial execution Both the concept of saga and its implementation are relatively simple, but they have the potential to improve performance significantly We analyze the various implementation issues related to sagas, including how they

the majority of other transactions either because it accesses many database objects, it has lengthy computations, it pauses for inputs from the users, or a combination of these factors Examples of LLTs are transactions to produce monthly account statements at a bank, transactions to process claims at an insurance company, and transactions to collect statistics over an entire database [Gray81a]

In most cases, LLTs present serious performance problems Since they are transactions, the system must execute them as atomic actions, thus preserving the consistency of the database [Date81a, Ullm82a] To make a transaction atomic, the system usually locks the objects accessed by the transaction until it commits, and this typically occurs at the end of the

https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf

Distributed Transaction Pattern: Saga Choreography



https://medium.com/swlh/microservices-architecture-what-is-saga-pattern-and-how-important-is-it-55f56cfedd6b

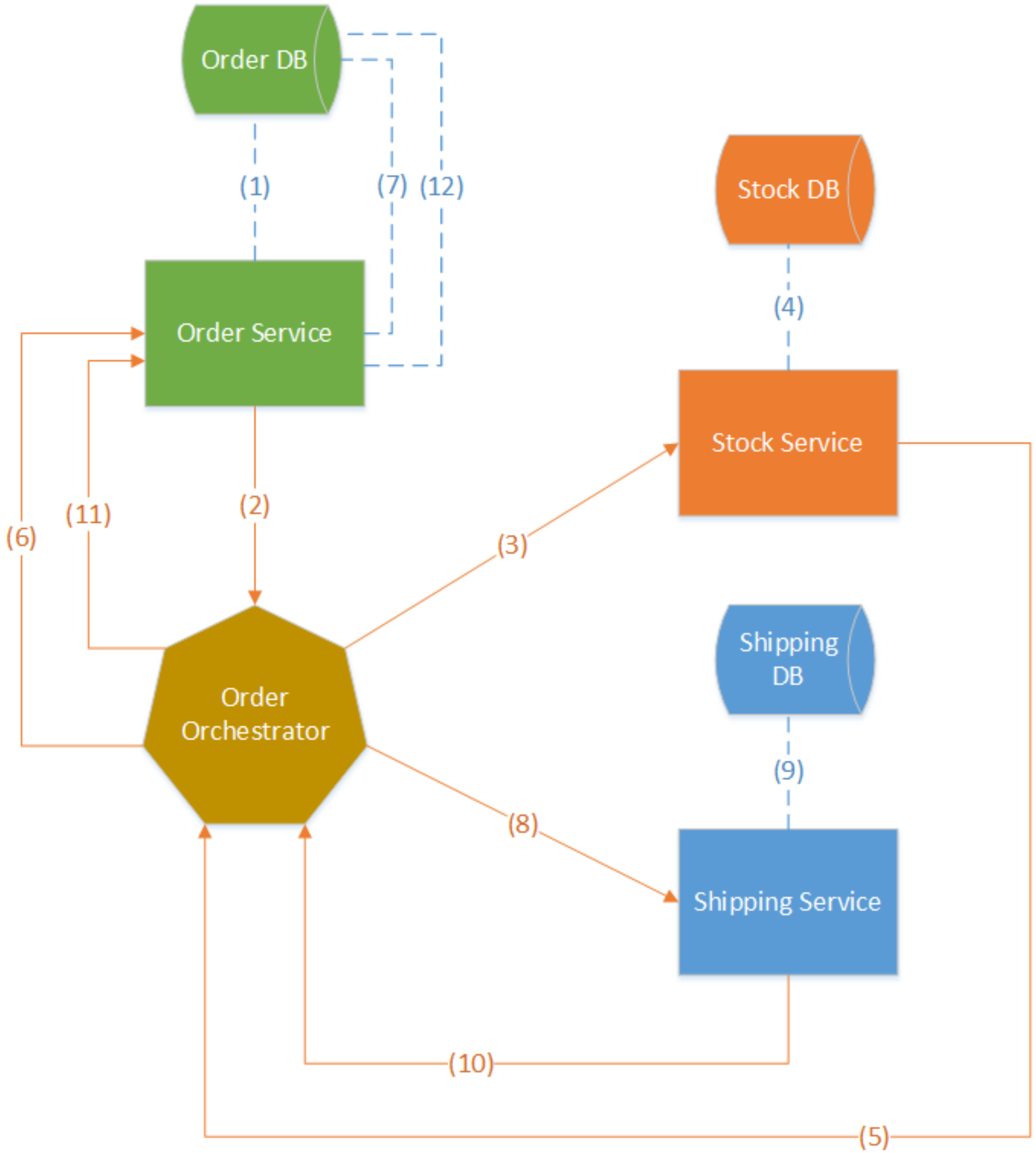
Saga Choreography Assumptions

- You create an ID for each transaction

- Each step can be rolled back if there is a failure

- All processes involved in transactions get notified if there is a failure

Variation: Saga Orchestration



https://medium.com/swlh/microservices-architecture-what-is-saga-pattern-and-how-important-is-it-55f56cfedd6b

Testing Saga Operations

Sagas need to handle non-failure conditions, such as insufficient stock to complete the order.

You therefore must identify, implement, and test all the rollback scenarios

Testing Saga Updates

Assumption: you need to test a new version of the Stock Service

Option 1: Stand up the entire subsystem with dummy databases

Option 2: Stand up only the new service, using stub services for the other parts of the subsystem

Final Thoughts

- Compare testing with engineering’s Verification, Validation, and Uncertainty Quantification (VVUQ) process for models
- Verification: no bugs in the algorithm
- Validation: works correctly for test cases
- Uncertainty Quantification: we know what we don’t know
- We’ll look more at performance testing and related topics (-> UQ) in the next lecture