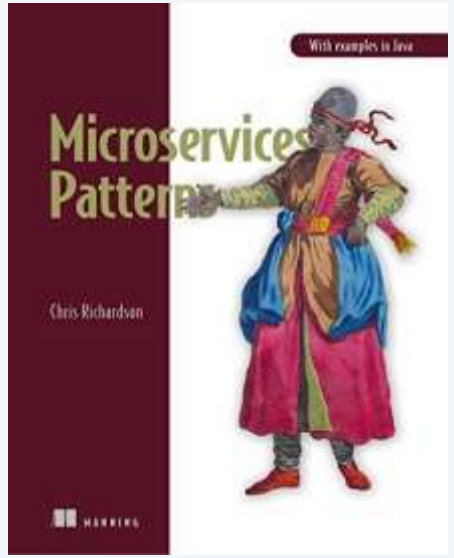
Testing Microservices

Objectives

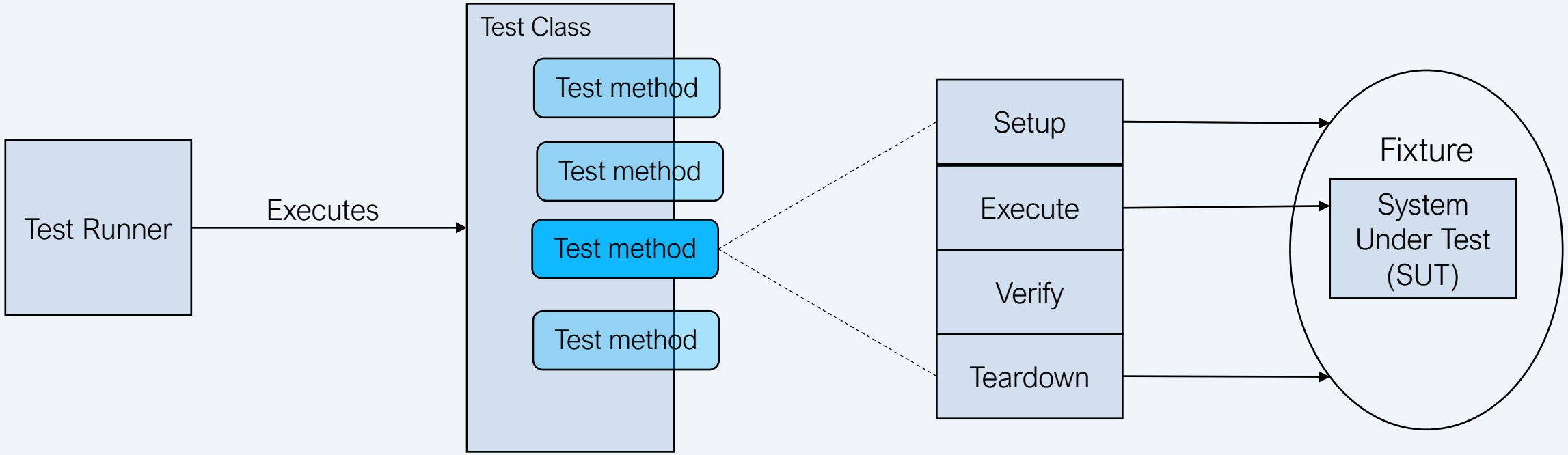
- Introduction to testing microservices
- Unit Testing
- Integration Tests
  - Contract testing
- Component Tests
  - User stories, Gherkin an Cucumber
- End-to-end tests



Introduction to Testing

- Automated testing is now essential to developing software
  - It is a vital link in the CI/CD chain
  - It is the way to having a short lead time for product releases and updates.
  - TDD or BDD is either strongly encouraged or mandated now
  - It is **not** just a QA thing, developers should be involved in writing tests from the very beginning of code development.

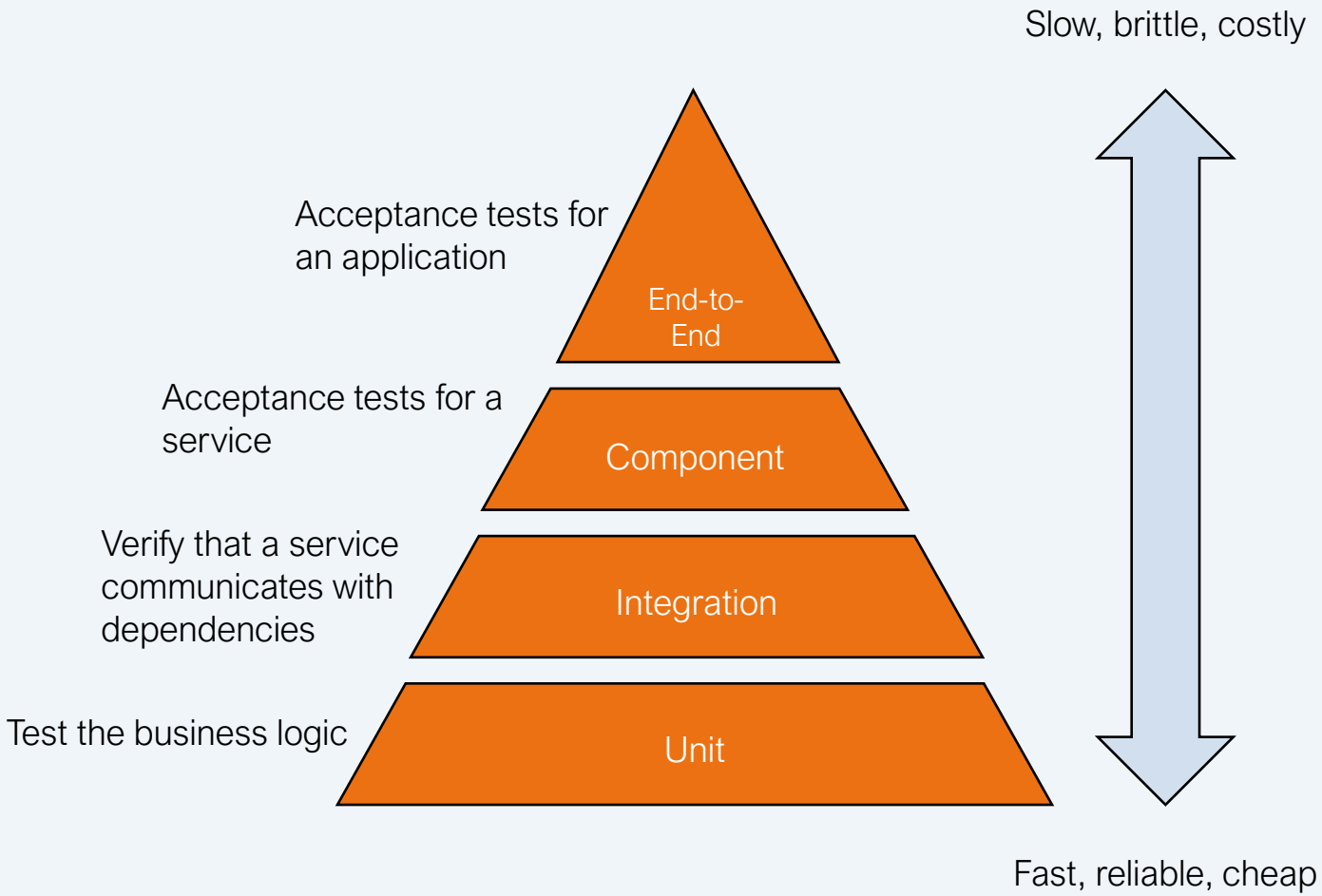
What does a test look like?



The Test Pyramid

- Microservices Testing Challenge:
  - Interprocess communication plays a central role
  - Teams are constantly developing their APIS and services.
  - It is essential a service is tested to verify interaction with clients and dependencies.

- The test pyramid should be used to focus test efforts for services
  - Fast, reliable, easy to write unit tests should be the majority of your tests.



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

Unit Testing

- Unit testing is the bread-and-butter of TDD.
  - For things like entity objects (like Order) and value objects (like Money) these are straightforward

- In a microservices environment it can be challenging because of the communication required.

Saga Unit Testing

- Need to have the “happy path”, as well as all the failure paths including rollback pathways.
- You *could* use a real message broker and database and stubs to simulate/mock the interacting services.
- Perhaps more effective to mock the classes that interact with the database and message broker (fast) so you don’t have to invoke a real database / message broker (slow)

Unit Testing Domain Services and Controllers

- The approach here is very similar to Sagas

- Essentially you are attempting to mock the first point of contact the service under test has outside of itself for any method invoked.

Integration Tests

- In this case we need to test that a particular service either produces or consumes the messages (either through API calls or asynchronous messaging) that are expected.

- This is more difficult that unit testing because it involves the interaction between two services.

- It is higher up the pyramid, so we want to do less of these – they are more expensive, slow and brittle!

Contract Testing

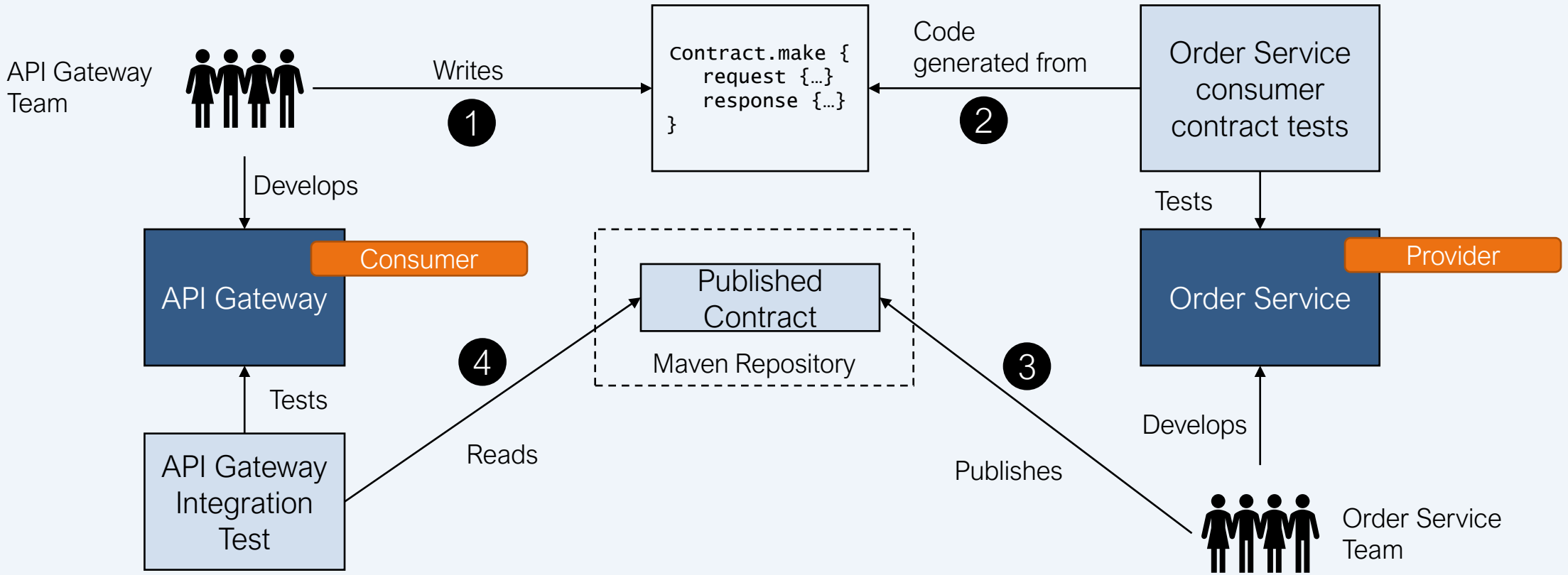
- An approach to communication/integration testing that verifies that a provider's API meets the consumer's expectations.

- Example: For a REST endpoint, the contract test verifies that the provider
  - Has the expected HTTP method and path
  - Accepts the expected headers
  - Accepts a request body
  - Returns a response with the expected status code, headers and body.

- In practice, the service consumers write a test specification of their expectations and submit it to the service providers (e.g. by a git pull request).
- The service providers then generate a suitable test for each consumer.

Example: Spring Cloud Contract Test

- The API Gateway consumes data from the Order Service API



Example: Spring Cloud Contract Test

- The contract script is written by the API Gateway team (the consumer) detailing the expected behaviour
  - In Spring Cloud Contract this is a file written using a Groovy domain specific language.
- Spring Cloud Contract automatically generates code
  - The provider team uses to test that the order service conforms to the expected behaviour.
  - Then it is published (packaged up as a JAR file) to provide the consumer with a “stub”, which is like a Mockito mock object.
- Result is the both the provider and consumer team can test for the expected behaviour without actually running the other service.

Contract tests

- Spring Cloud Contracts (https://cloud.spring.io/spring-cloud-contract) is Java based and can also test other communication methods – gRPC, or asynchronous messaging.

- The Pact family of frameworks (https://github.com/pact-foundation) is an alternative which supports a variety of languages.

Summary: Service Integration Contract Test Pattern

- Problem: how to easily test a service provides an API its clients expect?
- Forces: End to end testing is difficult, slow, brittle and expensive.
- Solution: A test suit for a service that is written by the developers of another service that consumes it. The test suite verifies that the service meets the consuming service's expectations.
- Resulting Context:
  - Benefits: Testing a service in isolation is easier, faster, more reliable and cheap.
  - Drawbacks: Only normal testing drawbacks – the test might not be comprehensive.
  - Issues: How to ensure the consumer provided tests are what they actually require?

Component tests (service acceptance tests)

- Now we want to verify that a service works as expected
  - Like a black box with behaviour verified through its API
- We could do this by deploying the whole application, but that is slow, brittle and expensive.
- Component testing verifies the behaviour of a service in isolation.

Defining acceptance tests

- Derived from user stories

  As a consumer of the Order Service
  I should be able to place an order

- We can expand this story into scenarios like

  Given a valid customer
  Given using a valid credit card
  Given the restaurant is accepting orders
  When I place an order for Chicken Vindaloo at The Carlton Curry House
  Then the order should be APPROVED
  And an OrderAuthorized event should be published

- The “Givens” correspond to test setup, the “When” maps to execute, the “Then/And” maps to verification.

Example: Cucumber and Gherkin

- You could take user scenarios defined as above and translate them into component tests yourself.
- Easier to use existing software.
- Gherkin is a domain specific language for writing executable acceptance tests.
- In Java, Gherkin combines with Cucumber to automatically generate the Java test code needed to run your tests.
- As before, you generally want to mock services and interfaces where possible.

End-to-end testing

- These are the top of the pyramid and it is good to have as few of these as possible.

- Once again, user stories are a good way to specify the tests, but now they will have multiple “when” statements (execute test) with corresponding “then” statement to verify the result.

Example: A Gherkin-based specification of a user journey

```
Feature: Place Revise and Cancel
  As a consumer of the Order Service
  I should be able to place, revise and cancel an order

  Scenario: Order created, revised and cancelled
    Given a valid consumer
    Given using a valid credit card
    Given the restaurant is accepting orders
    When I place an order for Chicken Vindaloo at The Carlton Curry House
      Then the order should be APPROVED
    Then the order total should be 16.33
    And when I revise the order by adding 2 vegetable samosas
      Then the order total should be 20.97
    And when I cancel the order
    Then the order should be CANCELLED
```

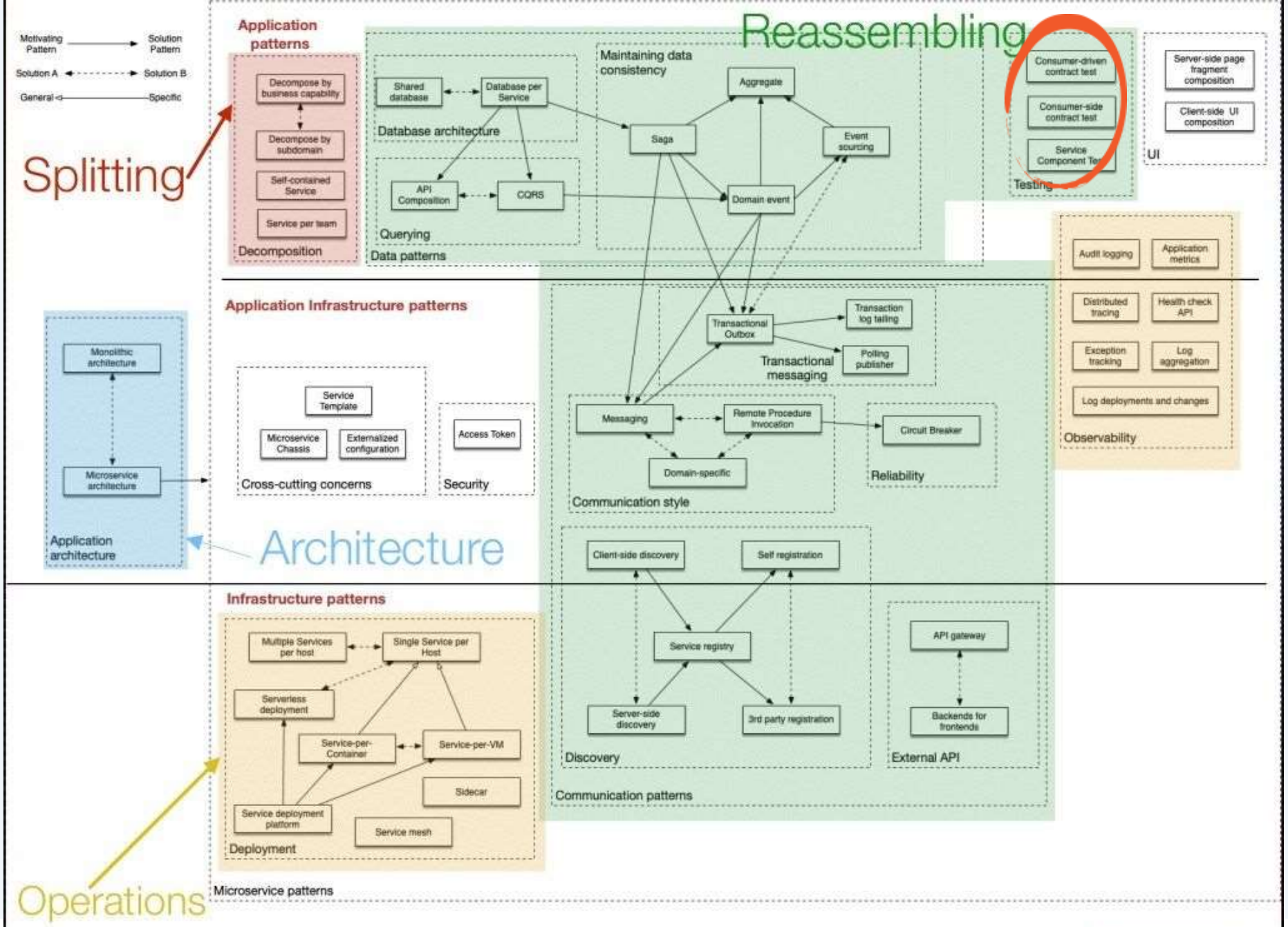
Create Order

Revise Order

Cancel Order

Running end-to-end tests

- Running an end-to-end test requires starting up all the application services and infrastructure.
- There are tools for this in Java, not sure about other languages!
- The Gradle or Maven Docker Compose Plugin provides a convenient way to run all the application services.
  - The tests are written in Gherkin and executed using Cucumber.
  - Maven Docker Compose plugin spins up all the services before the tests run.



Learn-Build-Assess Microservices http://adopt.microservices.io

Summary

- Introduction to testing microservices
- Unit Testing
- Integration Tests
  - Contract testing
- Component Tests
  - User stories, Gherkin an Cucumber
- End-to-end tests

Questions or Comments?