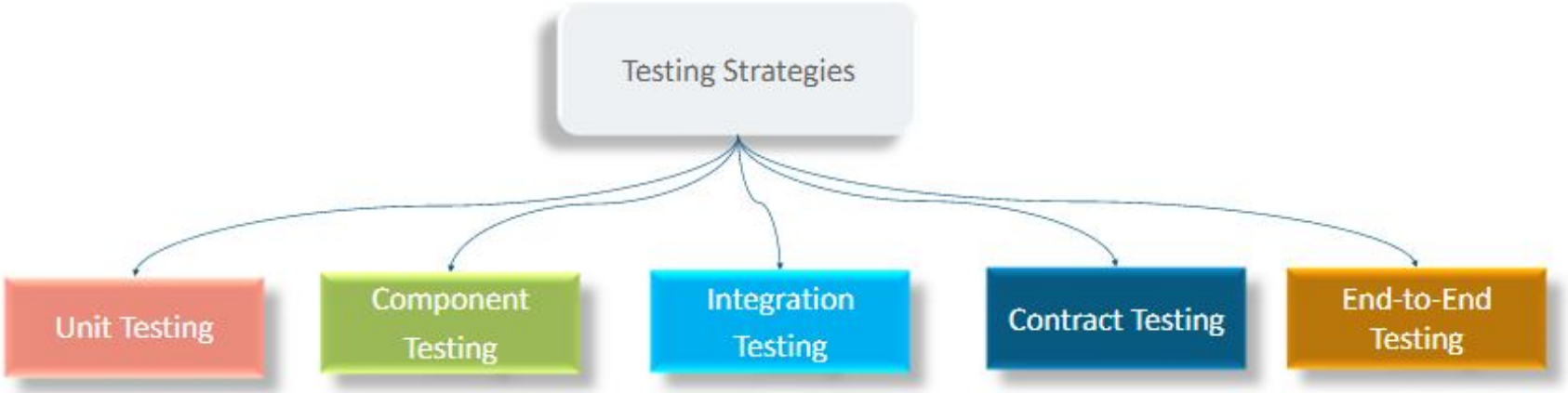
MICROSERVICES TESTING

Outline

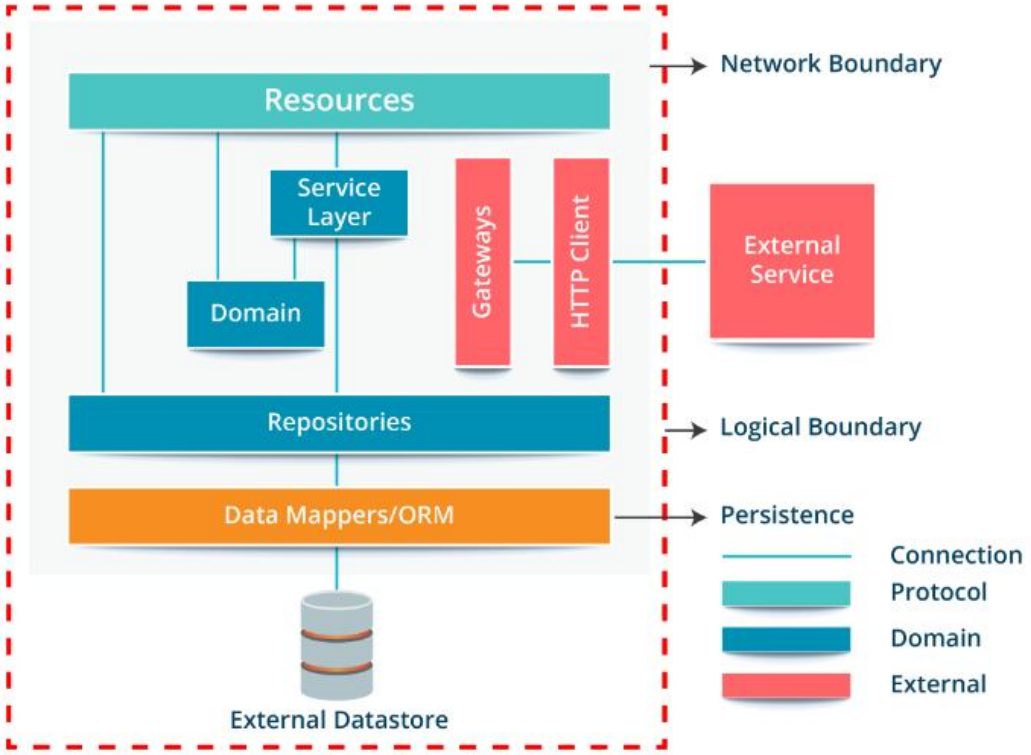
- Microservices Testing Strategies
- Unit Testing
- Integration Testing
- Component Testing
- Contract Testing
- End to End Testing

Testing Strategies



Microservice Eco System

- **Resource** – Resource, Service & Domain
- **Communication** – Gateway , Clients, External Services
- **Data Access** – External Database & Sources



Unit Testing

Microservices unit testing generally use the two styles given below. They are not competing styles and compliment each other while solving different testing problems.

Sociable Unit Testing

1. Focuses on testing the behavior of modules by observing changes in their state.
2. Treats the unit under test as a black box tested entirely through its interface, which plays key role while testing a function made of multiple microservices.

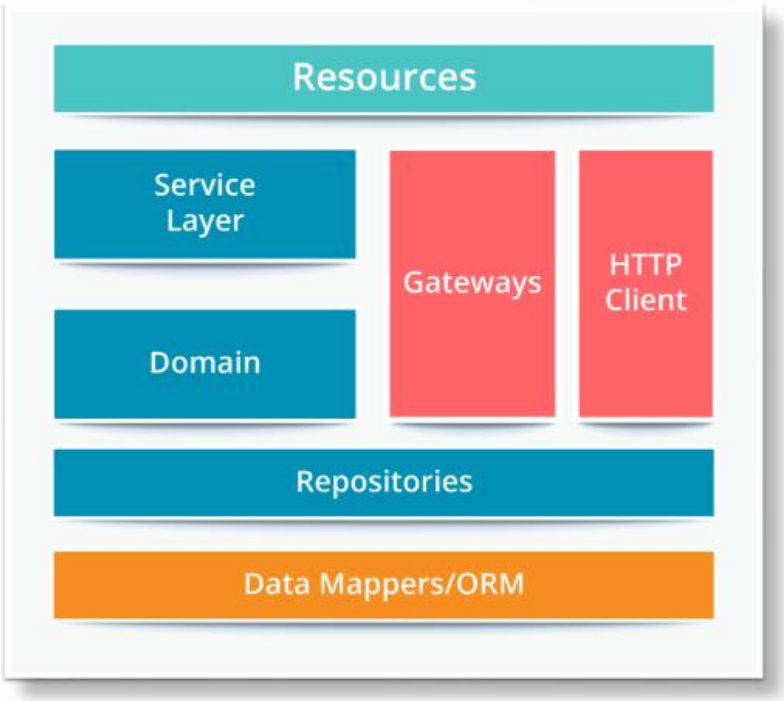
Solitary Unit Testing

1. Looks at the interactions and collaborations between an object and its dependencies, which are replaced by test doubles (Replacing Production object from Testing objects)

Overlay of Styles on the Ecosystem

Domain - logic involves complex calculations and a collection of state transitions which are highly state-based. There is little value in trying to isolate the units. Thus real domain objects should be used for all collaborators of the unit under test

Gateways - The purpose of unit tests here is to verify the logic used to produce requests or map responses from external dependencies and not to verify integrated communication. Doubles fit well here to ensure request and response are meeting the contract.



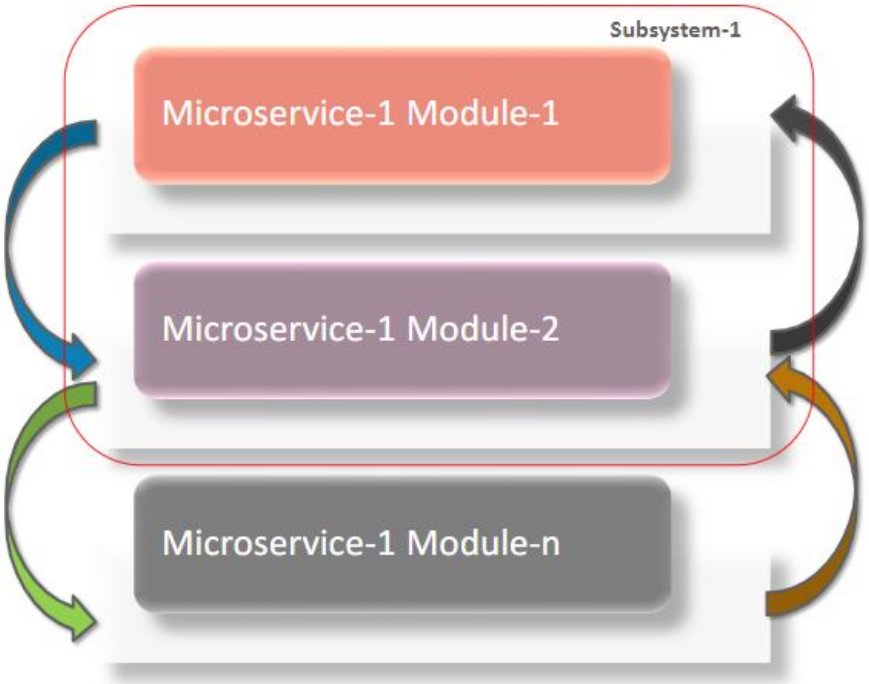
Unit Test – Where shall we stop?

As we achieve a fine grained microservices architecture, size of a service decreases leading to increase in plumbing and coordination logic to complex domain logic

There will be services, which will contain only plumbing and coordination logic such as adapters to different technologies or aggregators over other services.

When such situations arise the case for comprehensive unit testing may not pay off and one should look in next levels which provide more value, such as component testing.

Beyond Unit Test – Integration Test

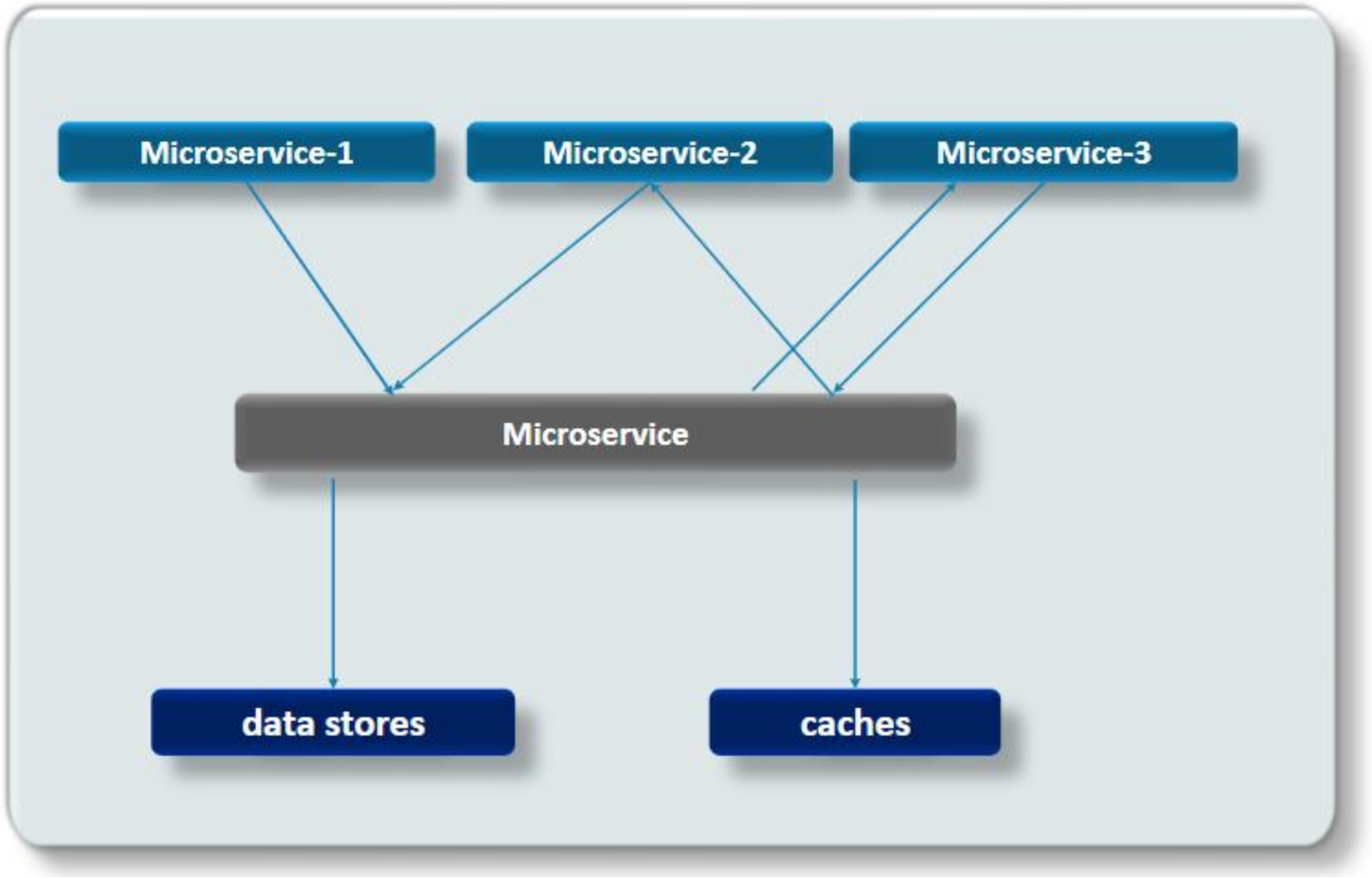


In order to complete testing a microservice, we need to ensure that each module correctly interacts with its collaborators. This requires more than just coarse-grained testing.

Group modules to test as a subsystem in order to verify that they collaborate as intended.

Exercise communication paths through the subsystem and ensure that assumptions each module has about how to interact with its peers is correct.

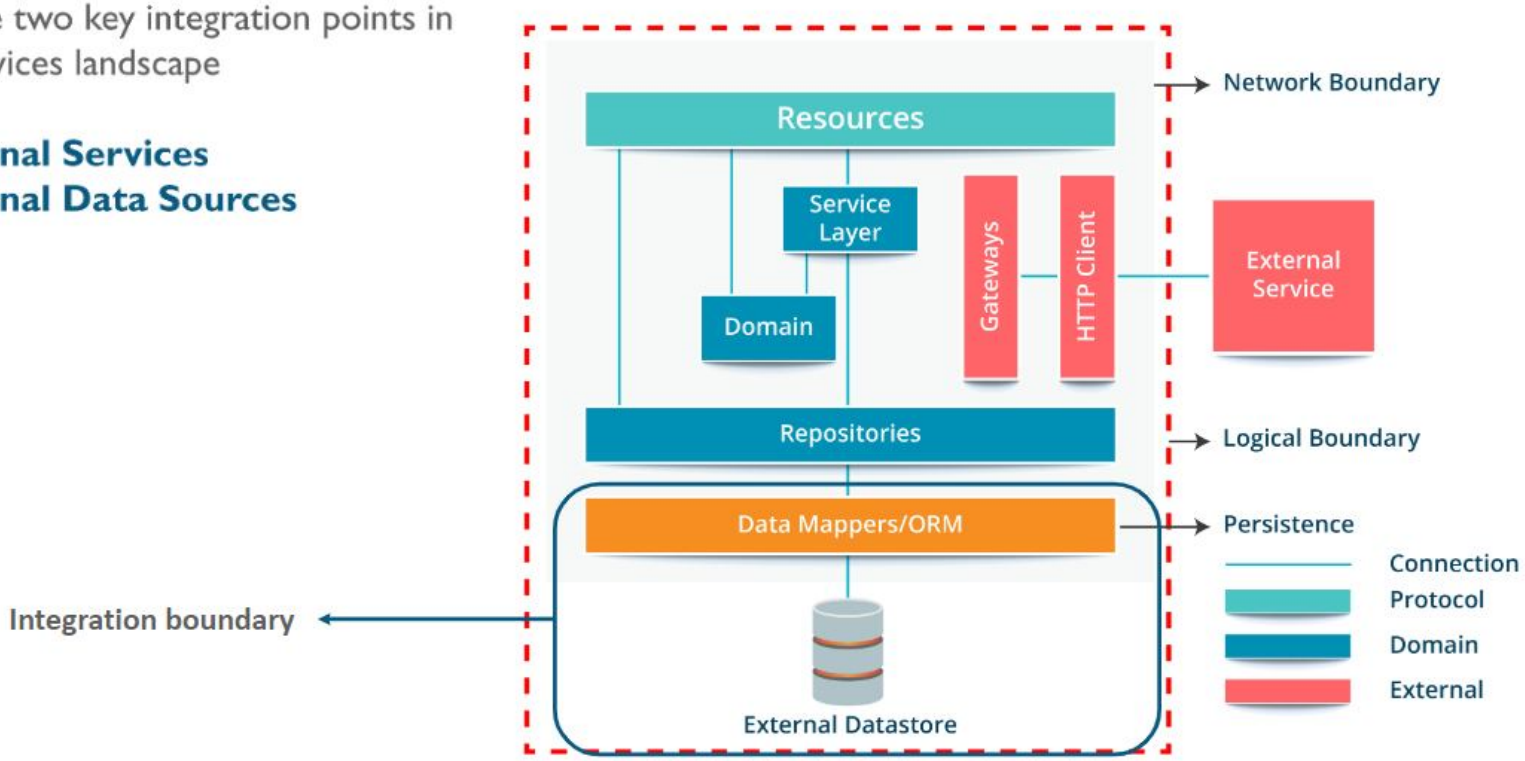
Microservices Integration



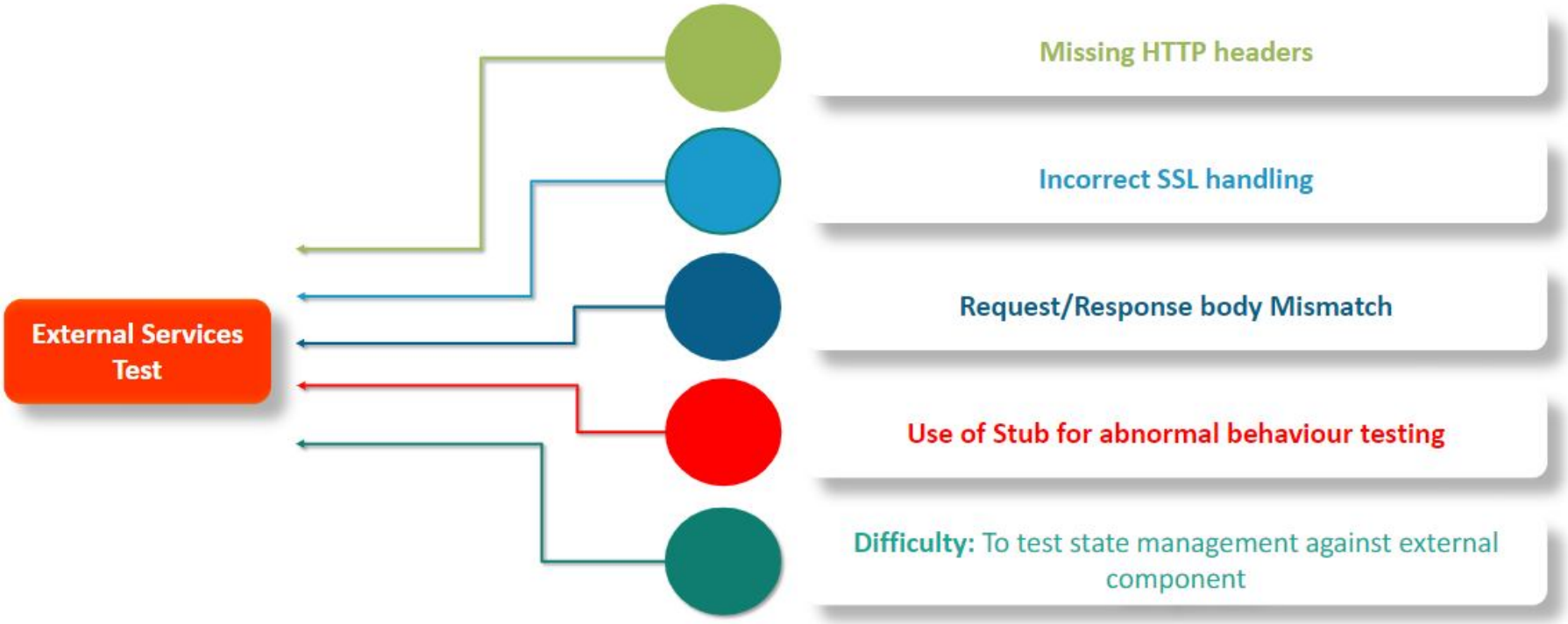
Integration Boundaries

There are two key integration points in microservices landscape

- **External Services**
- **External Data Sources**



External Services Test



External Data Source Test

Persisting the data in external data source provide assurances that the schema assumed by the code matches that which is available in the data source and is well integrated.

Structure tests such that the transactions close in between preconditions, actions and assertions to be sure that the data makes a full round trip.

Integration tests should attempt to verify that the integration modules handle the database failures gracefully.

Recap Unit and Integration Test

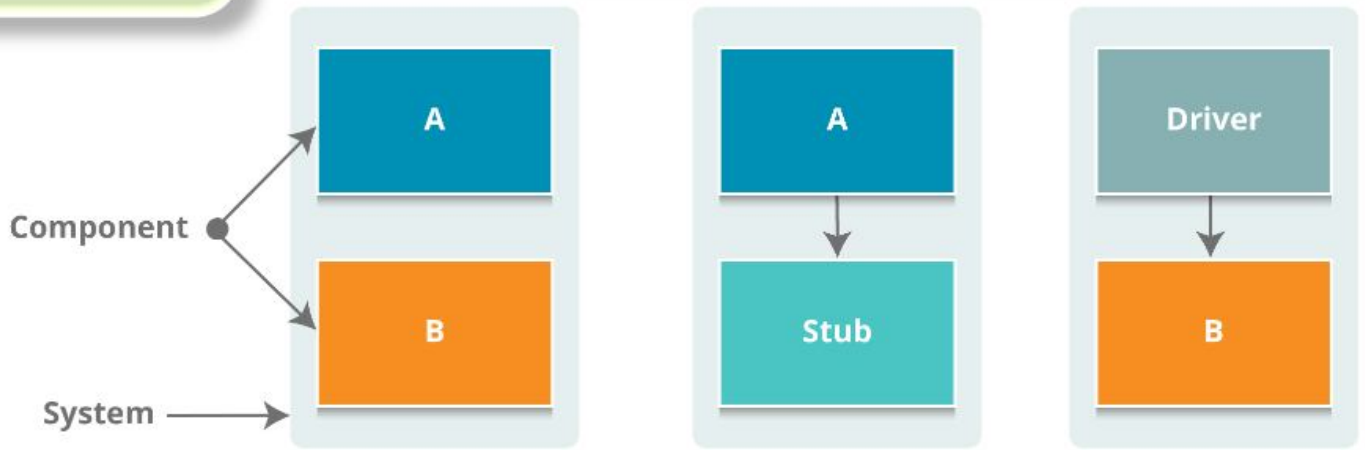
- Unit and integration testing gives us confidence in the correctness and completeness of the logic contained in the individual modules that make up the microservice.
- A more coarse grained suite of test is needed to make sure that the microservice works together as a whole to satisfy business requirements.
- While this can be achieved with fully integrated end-to-end tests, a more accurate test feedback is needed. A smaller test runtimes can be obtained by testing the microservice isolating it from its external dependencies.

Component Test

A ***component*** is any well-encapsulated, coherent and independently replaceable part of a larger system.

It limits the scope of the software under test to a portion of the system under test.

It manipulates the system through internal code interfaces.



Test doubles are used to isolate the code under test from other components.

Microservices Component Test

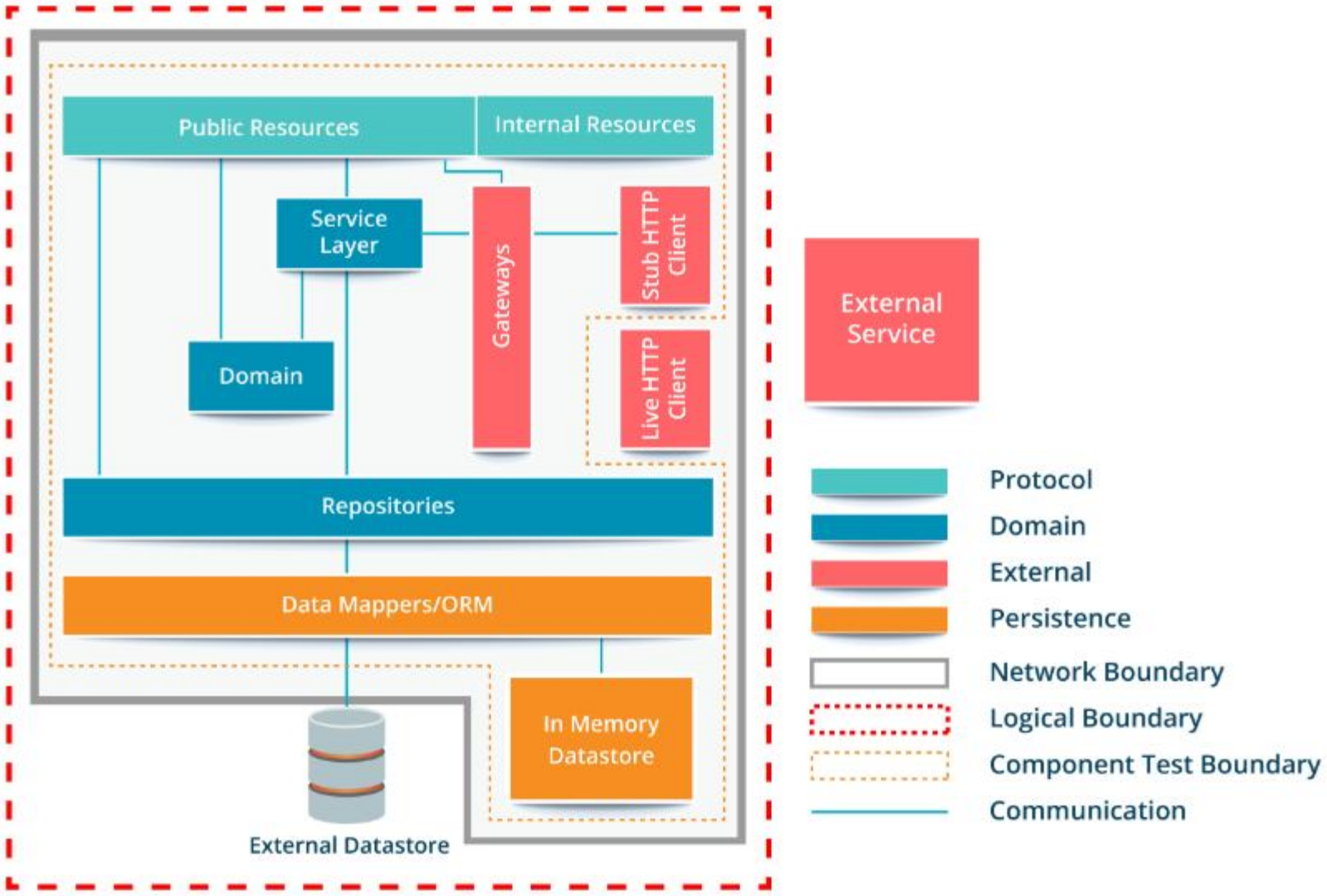
With respect to microservices architecture, the components are the services themselves.

Writing tests at this level of granularity makes the contract of the API driven through tests from the perspective of a consumer.

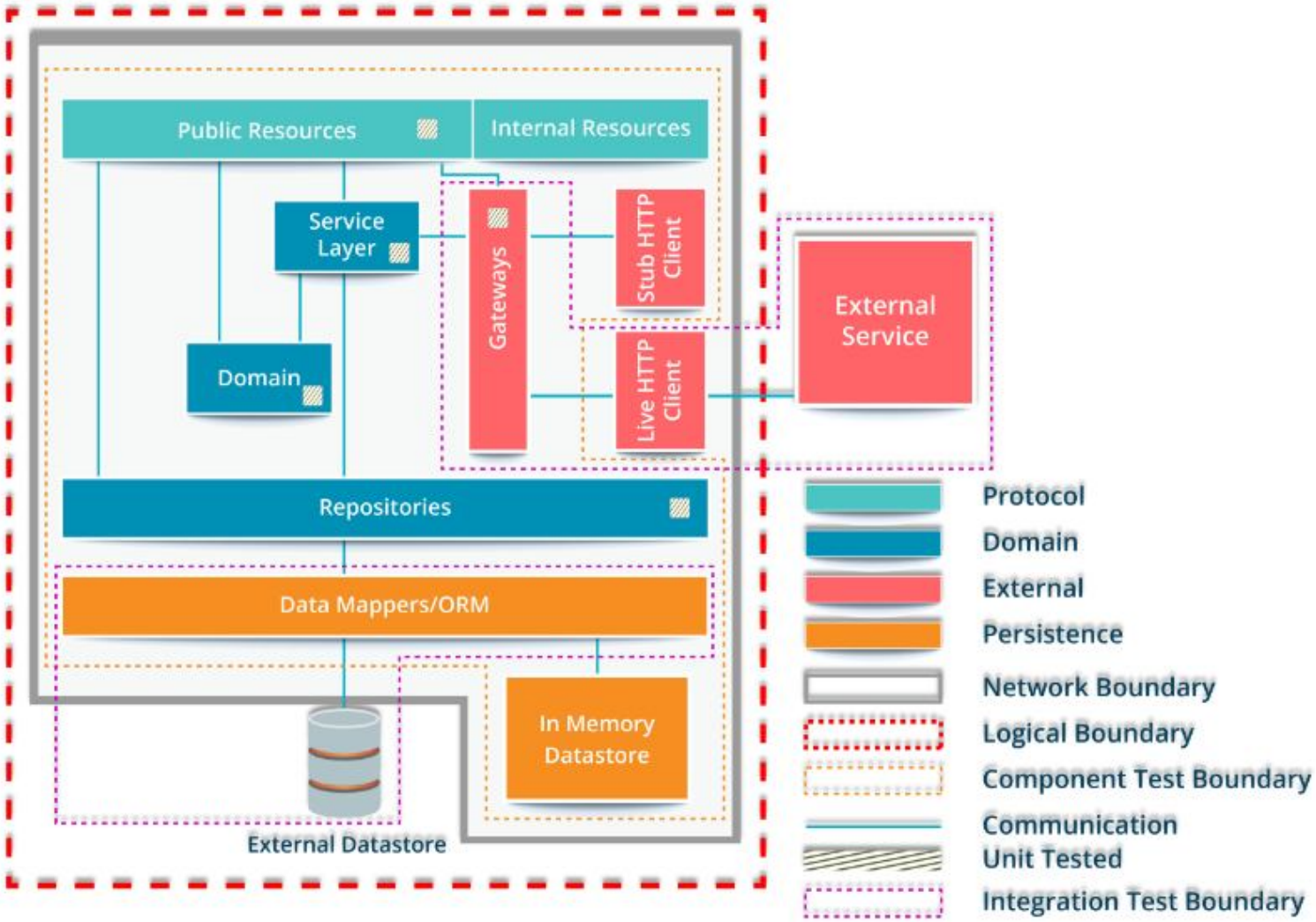
The implementation of such tests includes a number of options.

- Do the tests execute in the same process as the service or out of process through network ?
- Do the test doubles lie inside the service or externally ?
- Should a real data source be used or it should be replaced with an in-memory alternative?

Component Testing Boudaries



Boundaries Coverage



Business Process Coverage

Problem:

- How to ensure business functionality is achieved right?
- There are no tests that ensure external dependencies meet the contract expected of them **OR** that our collection of microservices collaborate correctly to provide end-to-end business flows.
- We have tested our system and ensured everything works as expected. However to ensure business process coverage we need to ensure that external services also comply with what they are expected to do or claim to do.

Solution:

This is achieved with contract testing *and is further explored with end to end testing.*

Contract Testing

Contract test is a test at the boundary of an external service which verifies that it meets the contract expected by a consuming service – Martin Fowler

Contract Testing *does* not test the behavior of the service deeply. *Rather, it tests that the inputs & outputs of service calls contain required attributes and that the response latency, throughput are within acceptable limits.*

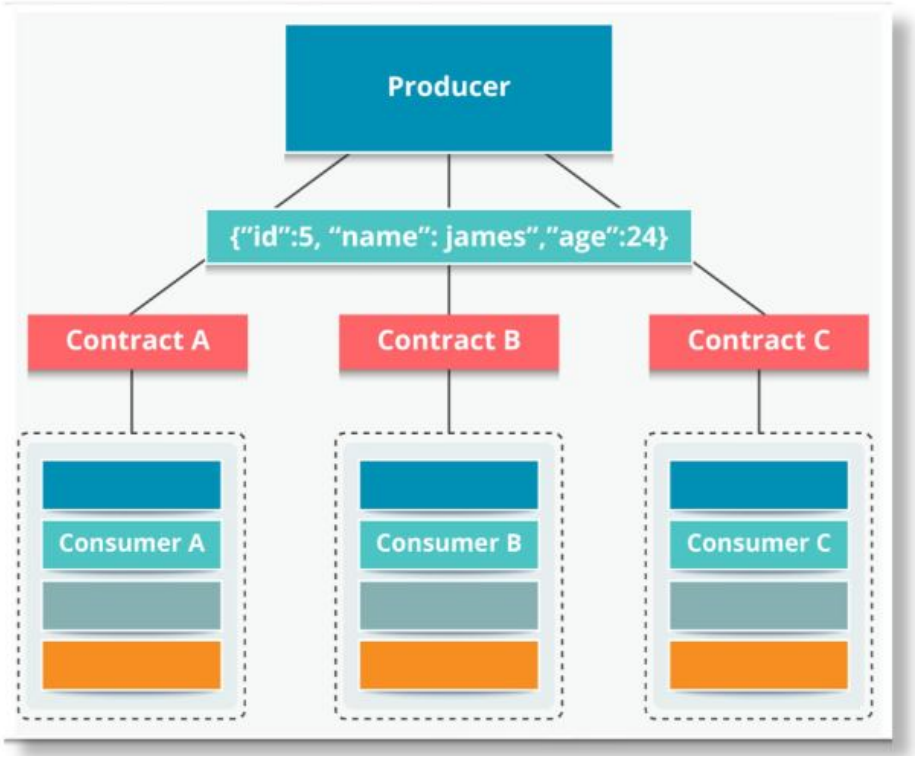
Contract Illustration

Consumer-A — A is not worried about Age

Consumer-B — B ignores Name

Consumer-C — C is interested in all

While testing we need to ensure based on consumer type the contract is validated and all scenarios are covered.



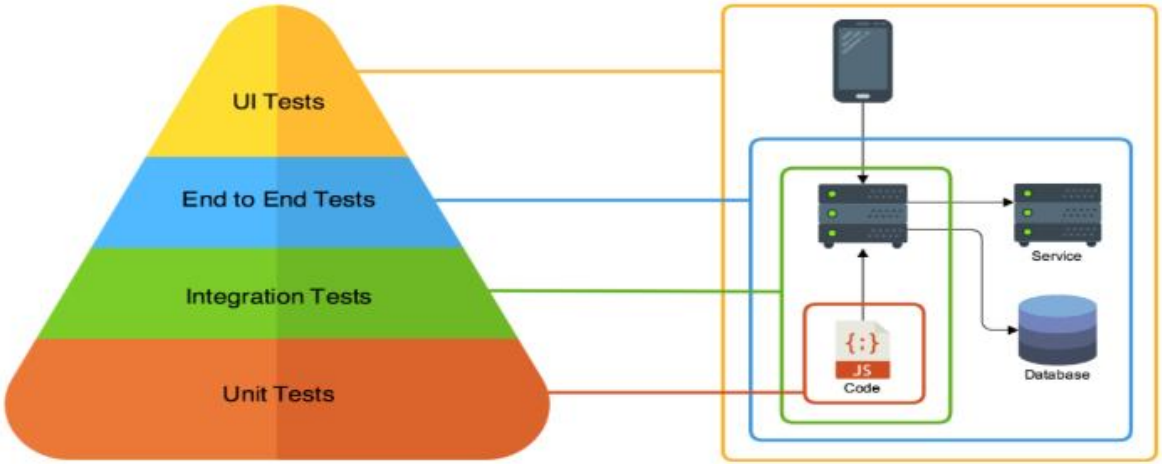
End to End Test

Verify that the system as a whole meets business goals irrespective of the component architecture in use.

1. *More business facing*
2. *Often uses domain Specific language*

End to End Microservices Test

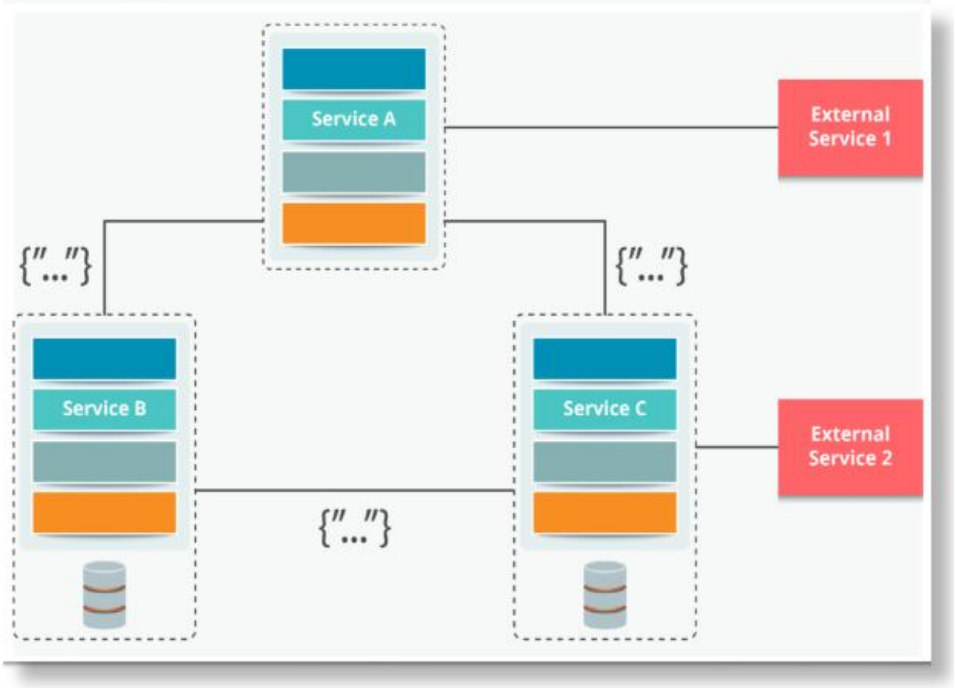
microservices architecture is built with multiple smaller parts that are choreographed dynamically, end-to-end tests provide value by adding coverage of the gaps between the services.



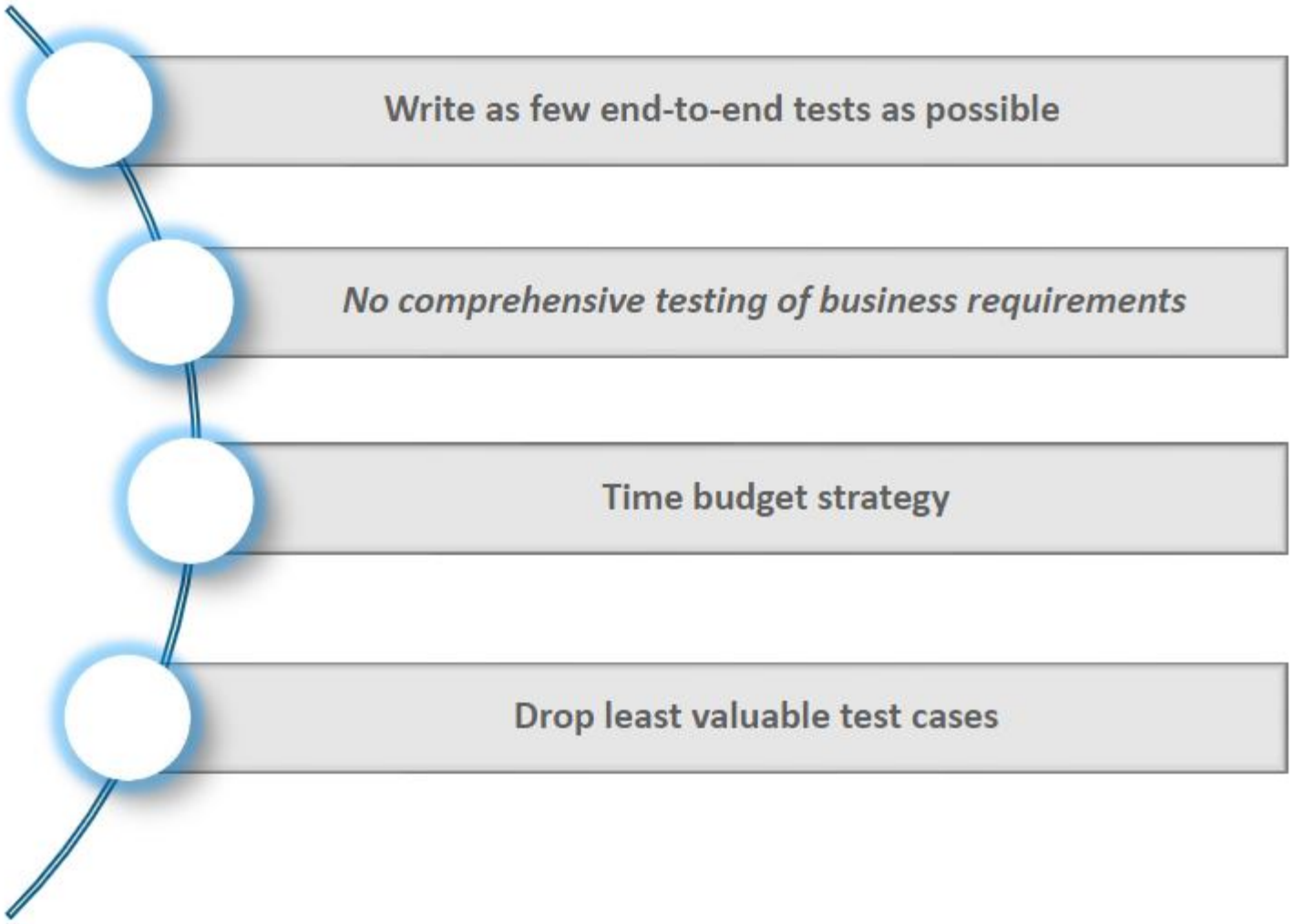
This ensures confidence in the correctness of messages passing between the services and verifies that the network infrastructure is correctly configured.

End to End Test Boundaries

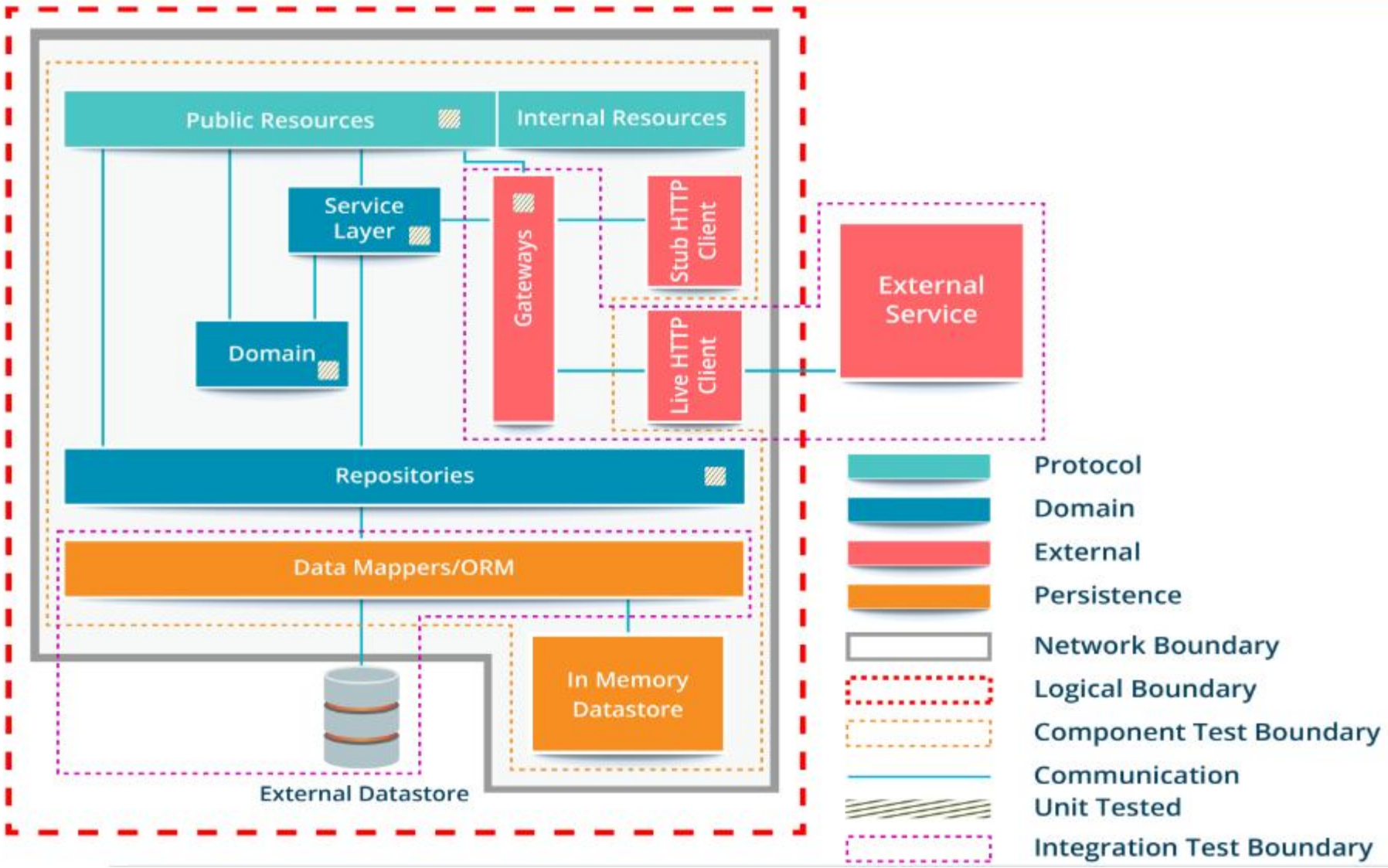
- Boundaries here are much wider than in other type of tests, making it complex to write these test cases in terms of possible scenarios.
- Another complexity arises as their services and components spread across different teams and projects, giving rise to communication channels and dependencies.
- Execution time could be higher due to span of functionality.



End to End Test Guidelines



Microservices Testing in Summary



Thank You!