Altimetrik

Shift-Left Testing Methodologies for Microservices

Challenges and Solutions

Author: Rajan Vishwa Ranjan

Agenda

INTRODUCTION 3

PRIMARY CHALLENGES IN THE TESTING OF MICROSERVICES. 4

1. NAVIGATING THE RAPID CURRENTS OF QA AND DEVELOPMENT IN AGILE 4
2. DELAYED FEEDBACK IN THE DEVELOPMENT CYCLE 4

SOLUTION 5

EVOLVING FROM PYRAMID TO DIAMOND STRATEGY 5

SHIFT LEFT TESTING STRATEGY 7

CONTRACT TESTING 7

CHOOSING THE RIGHT CONTRACT TESTING TOOL 7

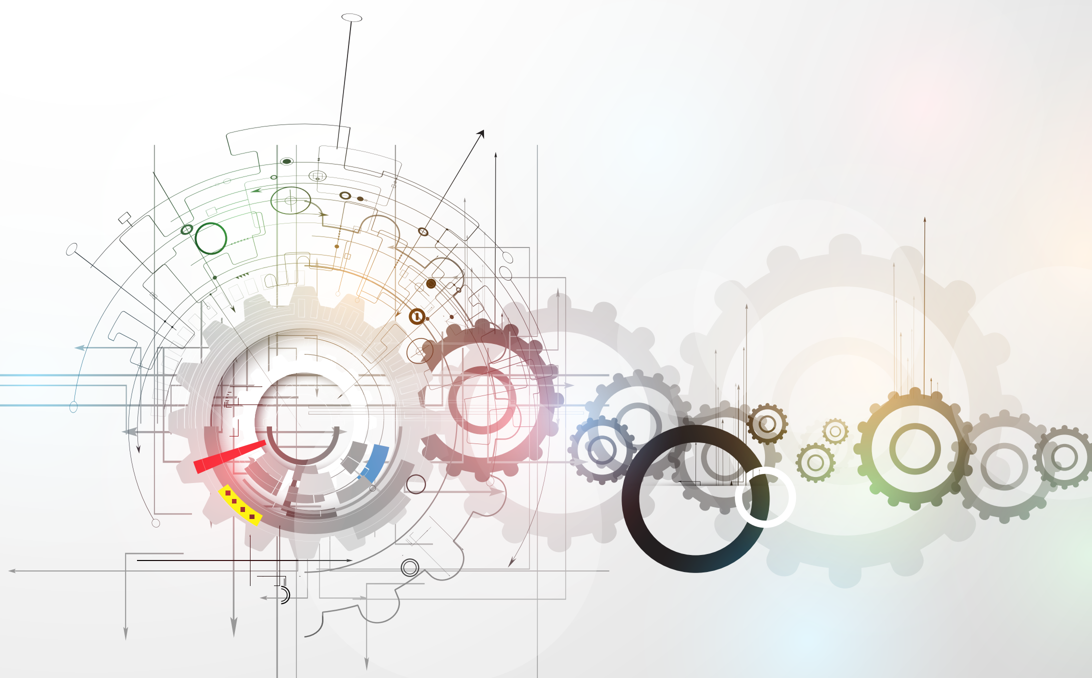
IMPLEMENTING PACT 8

PACT FLOW INTEGRATION WITH JENKINS 10

SERVICE VIRTUALIZATION 12

IMPLEMENTING SERVICE VIRTUALIZATION 12

CONCLUSION 16



Introduction

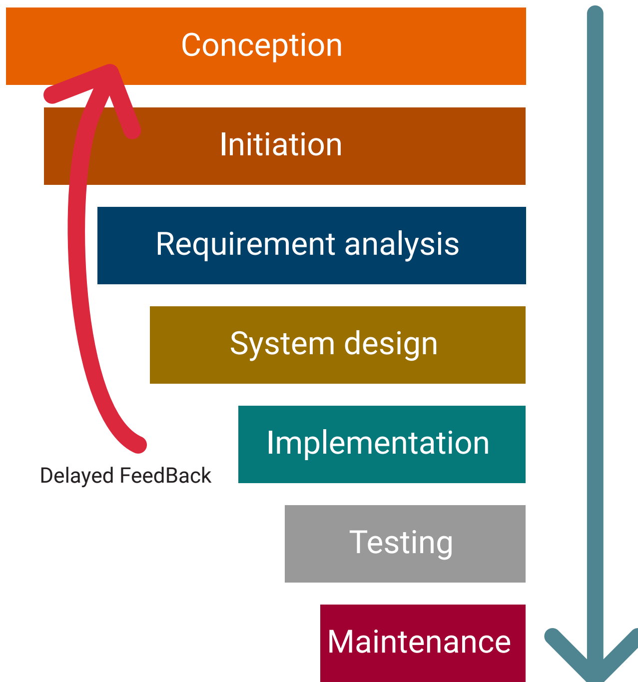
This white paper explores the enhancement of microservices quality through the integration of Shift-Left testing with Contract Testing and Service Virtualization. It highlights the importance of early and regular testing within the microservices architecture to address its inherent complexity and dynamic nature effectively.

The paper aims to provide in-depth insights into Shift-Left testing strategies and offer practical guidance for their implementation. The goal is to boost the development of resilient, high-quality microservices, promoting enhanced software reliability and quicker time-to-market.

By embracing these methodologies, organizations can substantially enhance their software quality assurance practices, resulting in improved business outcomes within a competitive digital landscape.

Primary Challenges in the testing of Microservices

Navigating the rapid currents of QA and Development in Agile



Agile methodologies and online software distribution have made automation and QA essential, diminishing the relevance of waterfall models.

Testing now occurs alongside development, covering everything from unit to end-to-end tests for instant feedback. However, development of microservices brings in a challenge as due to their interdependence and the complexity of testing across services and APIs which undergo constant upgrade in development phase.

Issues often emerge late in the process, forcing a return to earlier development stages or revealing problems during final integration tests.

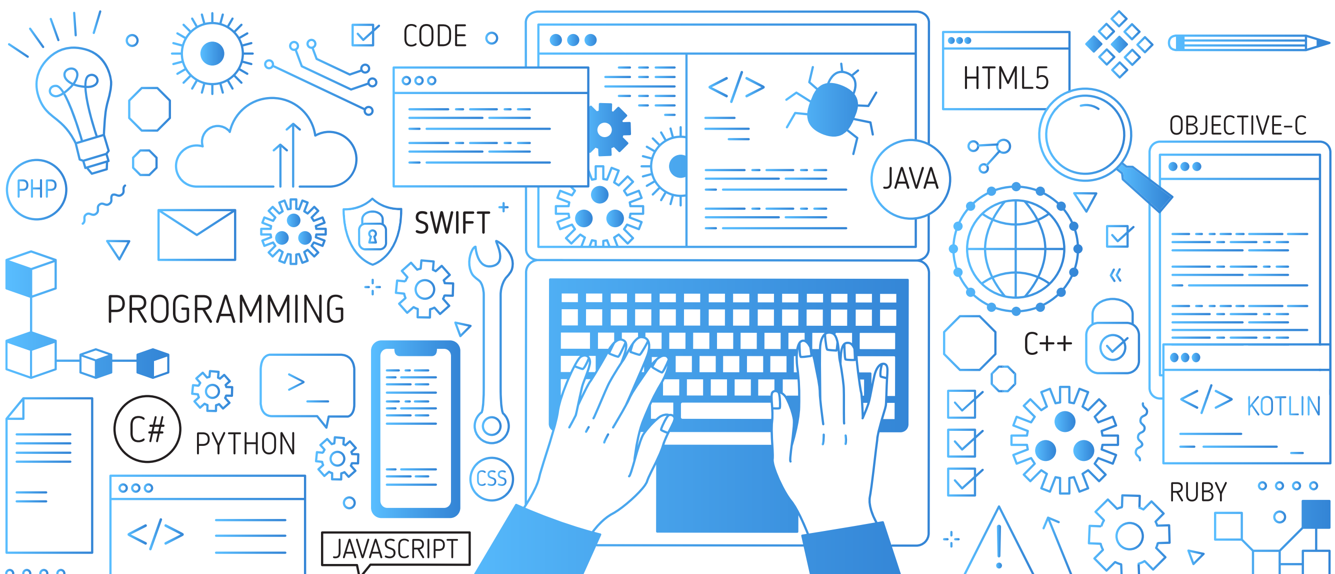
This results in delayed feedback and a drift back towards waterfall practices in agile testing environments.

Delayed feedback in the development cycle

Replicating a full cluster for testing can result in delays, with tests taking minutes to hours for feedback, deterring frequent testing by developers. Developers often skip integration tests, where services interact with the cluster, delaying these tests to later deployment stages.

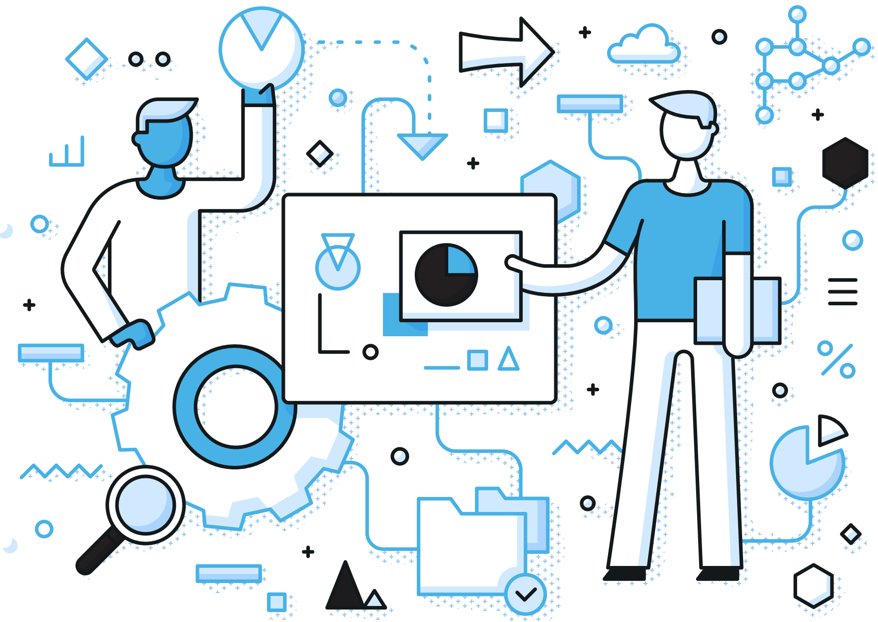
This delay leads to bugs being discovered late, reminiscent of waterfall methodologies, where bugs found by engineers from other teams cause inefficiencies in diagnosis, documentation, and resolution.

Consequently, the cost of fixing bugs increases significantly, illustrating a procedural challenge that impacts the efficiency and cost-effectiveness of the development process.



Solution

Let's delve into addressing the above challenges, which will serve as the central focus of this white paper.



Shift-left testing underscores the importance of integrating testing early in the development lifecycle, empowering developers to detect and rectify bugs when they are less costly to resolve. The diamond testing strategy integrates shift-left with later stages (shift-right testing), guaranteeing bug detection through continuous testing at every lifecycle stage, thereby minimizing errors and enhancing quality.

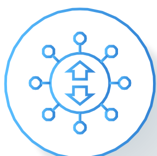
Furthermore, as we proceed, we will explore how contract testing and service virtualization contribute to testing microservices earlier in the development cycle.

Evolving from Pyramid to Diamond Strategy

Transitioning from the traditional test pyramid to a Diamond strategy for microservices testing signifies a shift in testing approach, recognizing the distributed nature of microservices architectures. This evolution introduces a more extensive integration testing layer, capturing the complex interactions between independently deployable services.

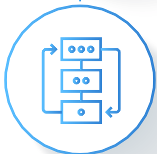
Consider an online retail platform consisting of multiple microservices: a Product Catalog Service, an Order Management Service, a Payment Processing Service, and a User Account Service. Each service is developed, deployed, and scaled independently, interacting through well-defined APIs.

In the traditional Pyramid approach, the focus of individual unit tests for services like the Product Service or Order Management Service often led to intricate mocks and test doubles. However, the return on investment (ROI) in terms of test confidence was relatively low due to several factors:



Isolated Testing:

Unit tests failed to capture issues arising from service interactions, resulting in integration bugs.



Limited End-to-End Coverage:

Comprehensive UI testing was challenging amidst rapid development cycles, overlooking certain user flow issues.

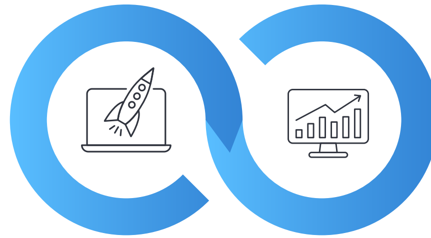
Service Dependencies:

Integration tests frequently encountered failures due to unavailable dependent services, thereby slowing down the development cycle.

To overcome these challenges, teams can adopt a Diamond strategy, specifically tailored for microservices, which emphasizes integration testing. This approach focuses on examining how the Payment Processing Service interacts with the Order Management and User Account Services, employing real API calls on the wire. This shift yields numerous benefits, including:

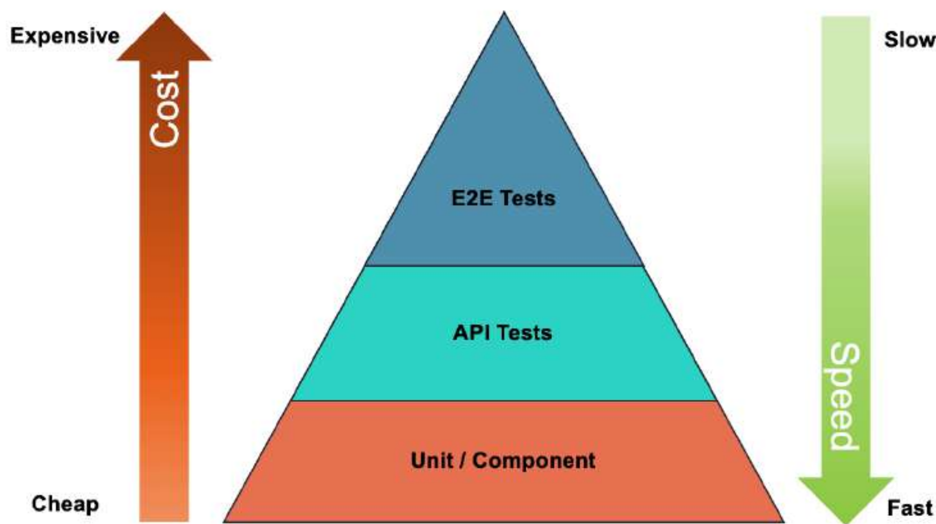
Faster Issue Identification

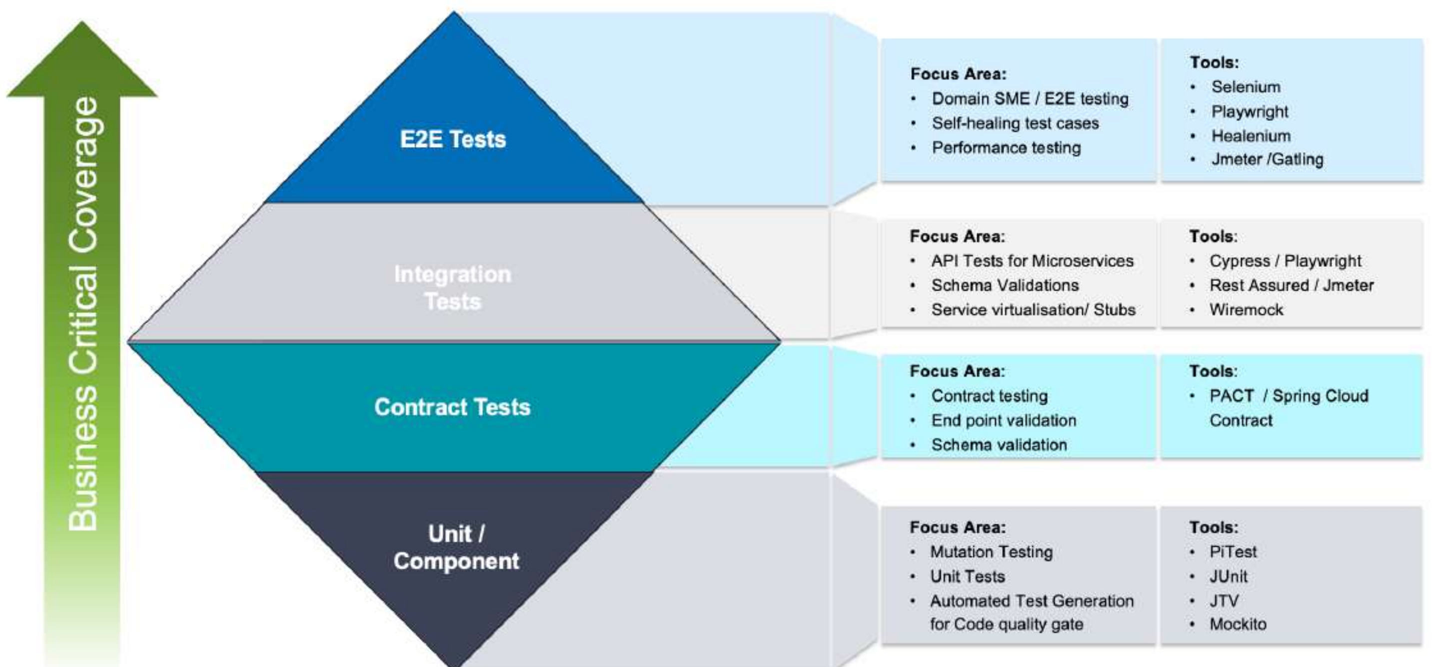
Expanded integration and contract testing swiftly pinpoint discrepancies in API contracts and data handling between services, thereby reducing debugging time.



Increased Deployment Confidence

With comprehensive testing spanning units, integrations, and contracts, the team can deploy updates more swiftly, knowing that critical workflows are thoroughly verified.





Shift Left Testing Strategy

This white paper focuses on two most important strategies that encompasses shift left testing and early bug detection.

1. Contract Testing
2. Service Virtualization

Let zoom in into both strategies.

Contract Testing

Contract testing is integral to left shift testing in microservices, facilitating early detection of integration issues. By validating service interactions against predefined contracts, it ensures compatibility and reliability across independently developed services. This proactive approach enables teams to identify and address potential problems at the development stage, reducing the likelihood of deployment failures.

As in the image below, API contracts such as schema, versioning, third party integration and backward compatibility testing are possible use cases, how contract driven testing helps to catch the bug early in the development cycle.

Choosing the Right Contract Testing Tool

There are several tools to choose from the market for contract testing. Based on the use cases and the testing requirement, the right tool should be chosen.

Here are some of the tools and its key features:

PACT

Overview: Pact is a prominent tool for consumer-driven contract testing, focusing on capturing the interactions between service consumers and providers in a contract file. This approach ensures that both sides understand and agree on how the APIs are used and respond.

Additional Details:

- **Consumer-Driven:** Enables consumer services to define their expectations in a Pact file, which acts as the contract.
- **Mock Services:** Pact mocks the provider services during testing, allowing consumers to test their interactions independently.

Spring Cloud Contract

Overview: Spring Cloud Contract is designed for developers working within the Spring ecosystem, providing tools to produce and consume contracts that verify REST and messaging interactions. It enables developers to work against service contracts, ensuring applications will work together without directly accessing each other's codebases.

Additional Details:

- **Contract Repository:** Contracts are stored in a central repository, making it easy for both providers and consumers to access and validate against them.
- **Automated Stub Generation:** Automatically generates stubs from the contracts, facilitating provider-independent testing for consumers.

WireMock

Overview: WireMock is an advanced tool for simulating HTTP-based APIs, allowing developers to mock web services in a flexible and realistic manner. It is not limited to contract testing but is widely used for this purpose due to its ability to accurately simulate the behaviour of external services.

Additional Details:

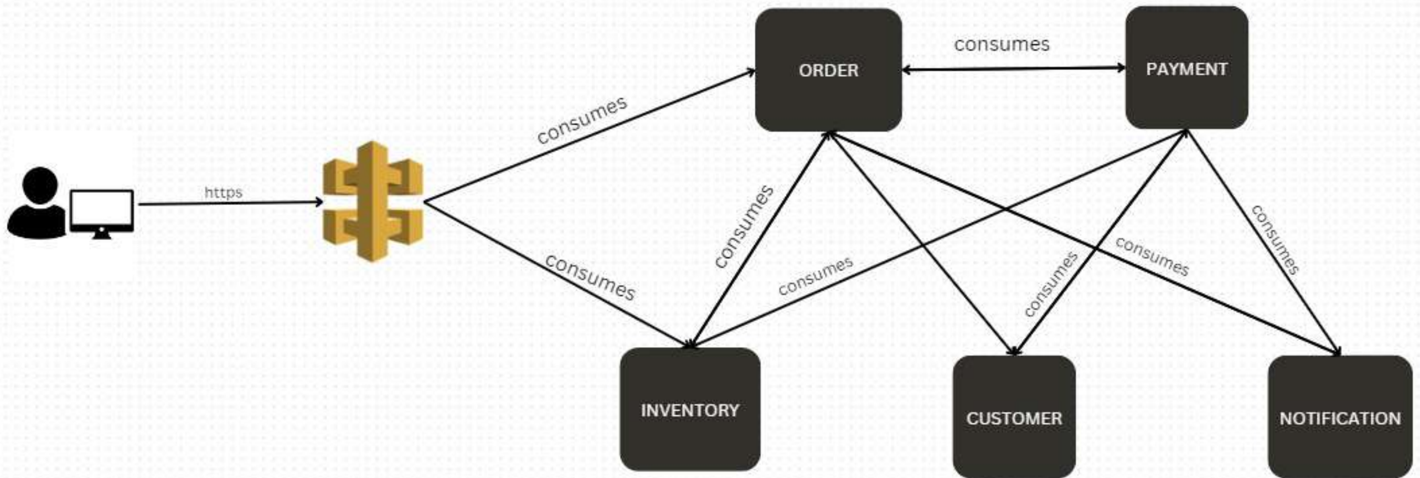
- **Simulation of HTTP Services:** WireMock can simulate any HTTP service, including REST and SOAP APIs, by mocking responses to requests.
- **Dynamic Response Creation:** Allows for dynamic creation of responses based on request parameters, enabling more realistic testing scenarios.

Implementing PACT

Example Scenario

Consider a microservices architecture, the Order Service and Payment Service often interact with several other services to fulfil their functionalities. For instance, alongside Order Service and Payment Service, there might be any number of microservices interacting with each other. Let's consider Inventory Service responsible for managing product availability, Customer Service handling user accounts and profiles, and Notification Service to notify users about order status.

The Order Service interacts with the Inventory Service to check product availability before placing an order. Once an order is placed, it communicates with the Payment Service to process the payment transaction. Additionally, the Order Service might notify the Customer Service to update user information or the Notification Service to inform users about the order status.

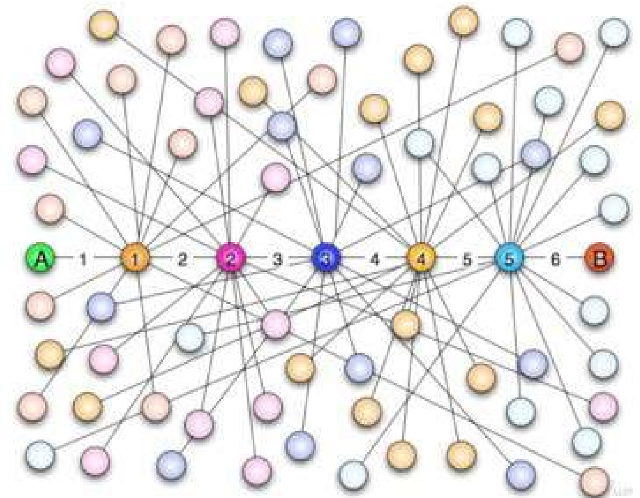


The diagram illustrates a network of microservices with five distinct services interacting with one another.

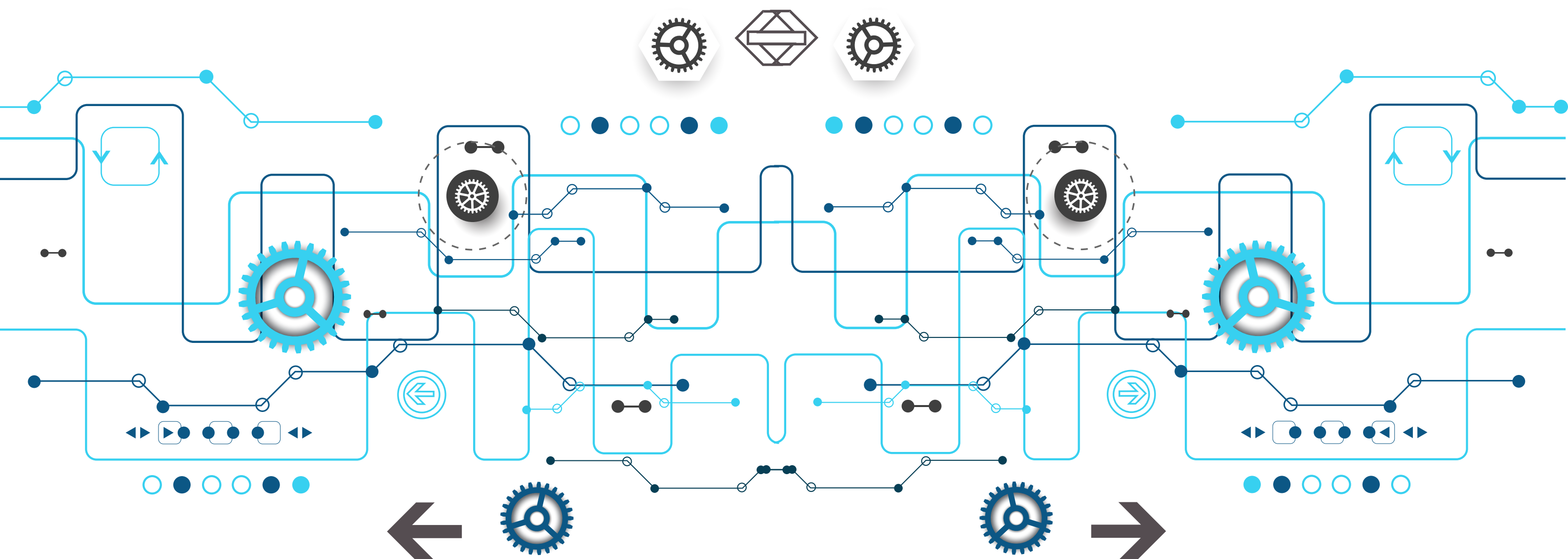
Each service has potential communication paths to the others. The number of potential communication paths in a network where each service can communicate with every other is not simply 5 multiplied by 5, which would suggest 25, but rather a more complex interconnection is at play.

If we generalize this, for n services where every service could potentially communicate with every other service, the number of possible communication channels could indeed become very large, specifically $n \times (n-1)$.

This is loosely referred to as dependency hell.

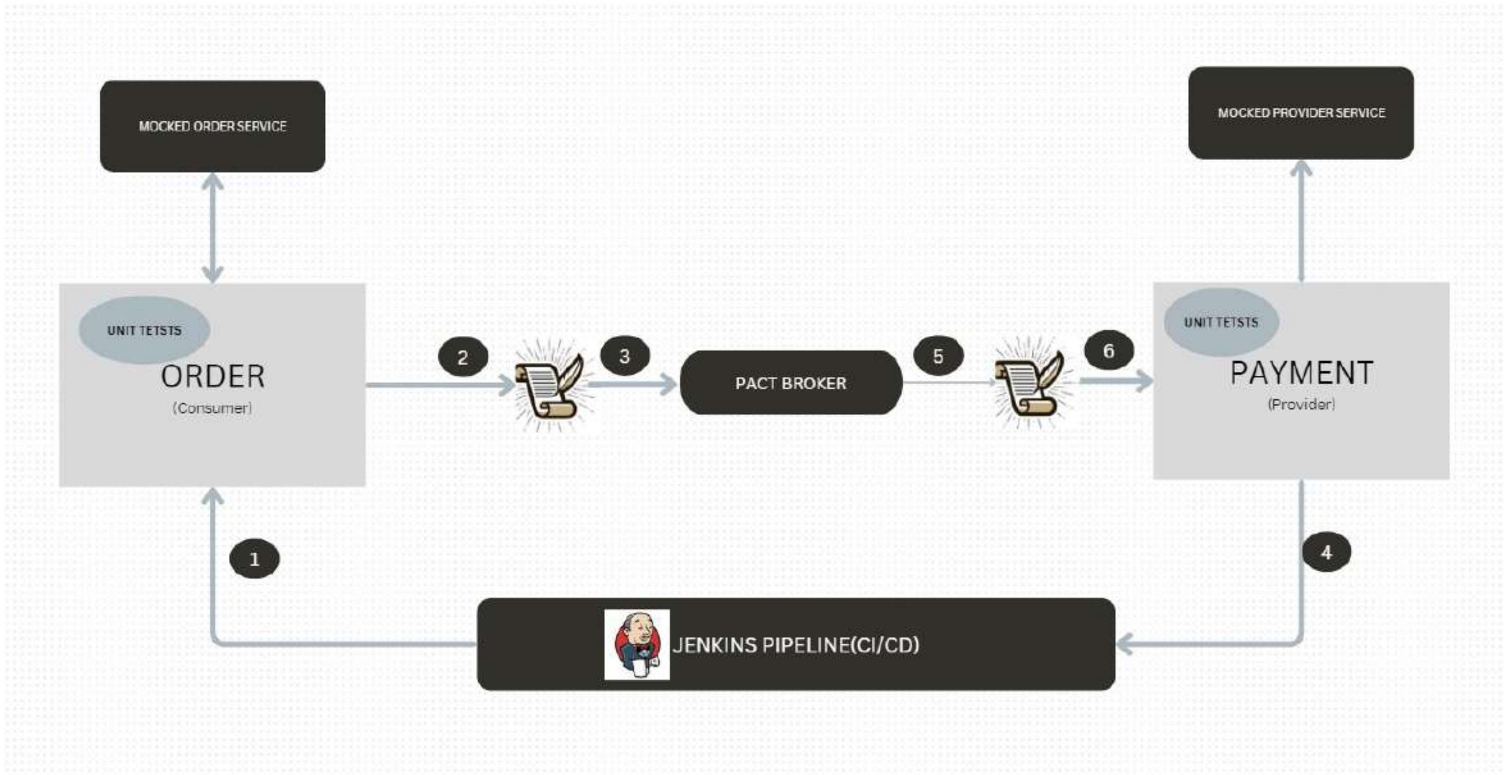


(Ref: MicroServices Martin Fowler, Netflix, Componentized Composable SOA



PACT Flow Integration with Jenkins

Integrating PACT within the Jenkins CI/CD pipeline facilitates consistent and automated contract testing. It triggers a sequence of events, ensuring that services adhere to their defined contracts and interaction patterns.



Step by Step PACT flow

Perquisite: A PACT broker facilitates the exchange of contracts between consumers and providers. It can be deployed as a Docker container with the following setup:.

```
docker run -d --name pact-broker \
          -e PACT_BROKER_DATABASE_ADAPTER=postgres \
          -e PACT_BROKER_DATABASE_NAME=pact_broker \
          -e PACT_BROKER_DATABASE
```

A Jenkins job is created to generate the contract file: This job is triggered when the code is merged in the remote repository. This pipeline is configured to trigger on a push event (git push could be replaced with a similar trigger for other SCM tools), executing a Maven command that runs the tests generating PACT contracts and optionally publishes them if the pactPublish property is set.

Interaction and Contract Generation by Order Service: The Order Service, acting as a consumer, interacts with a mocked version of the Payment Service to generate the contract. This process ensures that the consumer's expectations are clearly defined and documented. This code defines the expectations of the Order Service for a payment processing request, including the expected request path, method, body, and response.

```
pipeline {
    agent any
    triggers {
        pollSCM('H/5 * * * *')
    }
    stages {
        stage('Generate Contract') {
            steps {
                script {
                    // Code to merge or verify merge is completed
                    sh 'mvn clean test -Dpact.verifier.publishResults=true'
                }
            }
        }
    }
}
```

```java
@Pact(provider = "OrderService", consumer = "PaymentService")
public RequestResponsePact createPact(PactDslWithProvider builder)
{   return builder
        .uponReceiving("a request for order")
            .path("/order")
            .method("GET")
        .willRespondWith()
            .status(200)
            .body(new PactDslJsonBody()
                .integerType("id", 123)
                .stringType("status", "completed"))
        .toPact();
}
```

Publishing the Contract to the PACT Broker: Once the contract is generated, it is published to the PACT Broker, a central repository where contracts are stored and shared between the consumer and provider services.

```
mvn pact:publish -Dpact.broker.url=http://pact-broker-url -Dpact.consumer.version=1.0.0
```

Triggering Provider Unit Tests with a Jenkins Job: A separate Jenkins job is set up to trigger the Payment Service's provider unit tests. These tests interact with the mocked provider to verify that the service can fulfil the contract.

```groovy
pipeline {
    agent any
    triggers {
        pollSCM('H/5 * * * *')
    }
    stages {
        stage('Provider Test') {
            steps {
                script {
                    // Code to trigger provider tests
                    sh 'mvn clean test -Dpact.provider.tags=PROD'
                }
            }
        }
    }
}
```

Receiving the Contract from the PACT Broker: The Payment Service, as the provider, retrieves the contract file from the PACT Broker. This is typically handled automatically by the PACT library when the provider tests are run.

Verifying the Contract by the Payment Service: This code configures the Payment Service to verify interactions defined in the PACT file retrieved from the specified PACT Broker URL, ensuring that the service can handle requests as expected by the Order Service

```java
    // You would typically set up your service and its state here
    @BeforeEach
    void before(PactVerificationContext context) {
        context.setTarget(new HttpTestTarget("localhost", 8080, "/"));
    }

    @State("an order with ID 123 exists and is ready for payment")
    public void toOrderReadyForPaymentState() {
        // Set up your provider state here
        // For example, prepare your service with an order with ID 123 that has a status of 'completed'
    }

    @TestTemplate
    @ExtendWith(PactVerificationInvocationContextProvider.class)
    void pactVerificationTestTemplate(PactVerificationContext context) {
        context.verifyInteraction();
    }
}
```

Service Virtualization

Service virtualization involves creating lightweight, scalable, and deployable simulations of services. These virtual services mimic the real services' APIs and responses, enabling teams to test the interactions between microservices without the need for all services to be fully implemented or available.

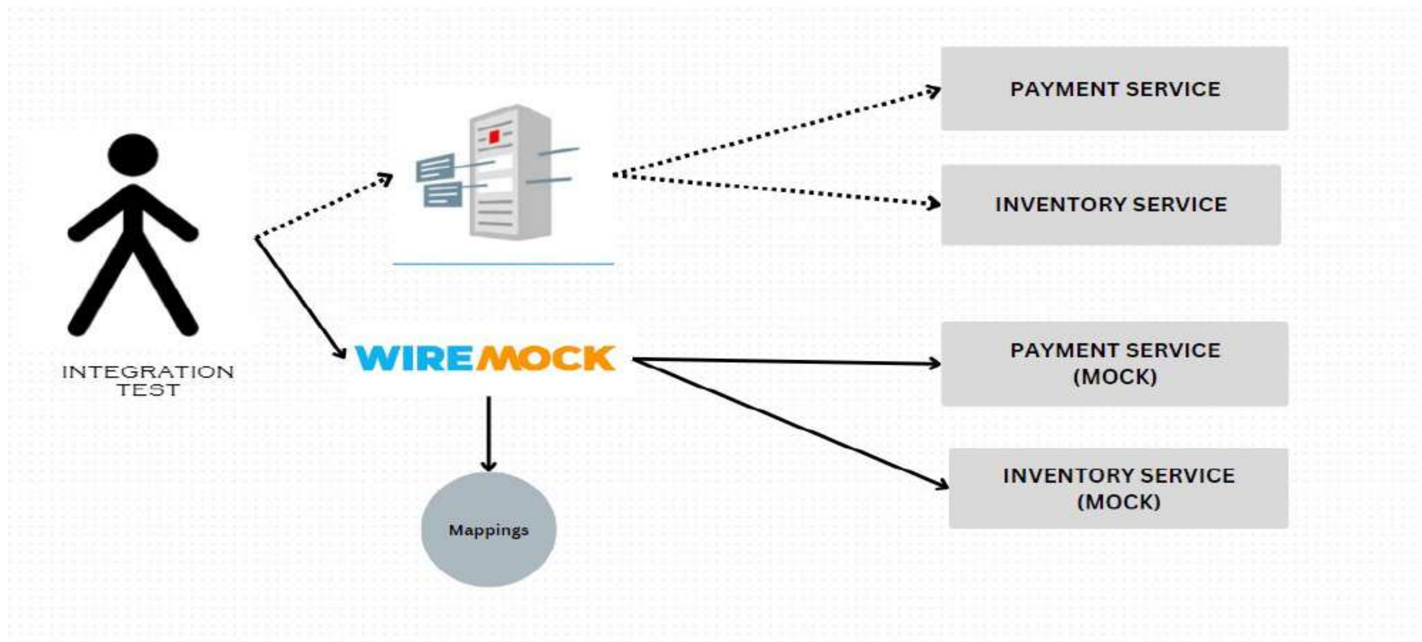
Benefits of Service Virtualization

- **Reduced Testing Time and Costs:** By removing dependencies on real service implementations, testing can proceed in parallel with development.
- **Increased Test Coverage and Quality:** Enables testing of edge cases and failure modes that are difficult to replicate with real services.
- **Enhanced Agility:** Supports a shift-left approach, allowing testing to occur earlier in the development cycle.

Implementing Service Virtualization

Strategy and Planning

- **Identify Dependent Services:** Map out the microservices architecture to identify which services depend on others.
- **Determine Virtualization Needs:** Assess which services would benefit most from virtualization based on development timelines and testing requirements.



Tools and Setup: Use WireMock, a flexible library for stubbing and mocking web services. It allows simulation of responses of the Payment Service and Inventory Service.

Step-by-Step Implementation

Virtualizing the Payment Service: First, create a virtual Payment Service file that approves payment requests. Define the below Json mapping that listens for POST requests to /api/payments and responds with a JSON payload indicating success.

```
{
  "request": {
    "method": "POST",
    "urlPath": "/api/payments"
  },
  "response": {
    "status": 200,
    "body": "{\"status\":\"Approved\",\"transactionId\":\"12345\"}",
    "headers": {
      "Content-Type": "application/json"
    }
  }
}
```

Virtualizing the Inventory Service: Next, virtualize the Inventory Service to confirm item availability. The below Json mapping listens for Get /api/inventory/check and responds with a Json Payload indicating success.

```
{
  "request": {
    "method": "GET",
    "urlPath": "/api/inventory/check"
  },
  "response": {
    "status": 200,
    "body": "{\"itemId\":\"A1\",\"status\":\"Available\"}",
    "headers": {
      "Content-Type": "application/json"
    }
  }
}
```

Create the setup in the test file that starts the wiremock server for Payments and Inventory services.

```
@BeforeEach
void setUp() {
    // Initialize WireMock servers for Payment and Inventory Services
    mockPaymentService = new WireMockServer(options().port(8081));
    mockInventoryService = new WireMockServer(options().port(8082));
    mockPaymentService.start();
    mockInventoryService.start();

    // Configure WireMock for Payment Service to ss status
    String paymentServiceResponseJson = "{\"status\":\"success\"}";
    mockPaymentService.stubFor(post(urlEqualTo("/api/Payments"))
        .willReturn(aResponse()
            .withHeader("Content-Type", "application/json")
            .withStatus(200)
            .withBody(paymentServiceResponseJson)));

    // Configure WireMock for Inventory Service to confirm stock availability
    String inventoryServiceResponseJson = "{\"available\": true}";
    mockInventoryService.stubFor(get(urlEqualTo("/api/inventory/check"))
        .willReturn(aResponse()
            .withHeader("Content-Type", "application/json")
            .withStatus(200)
            .withBody(inventoryServiceResponseJson)));

    // Initialize OrderService with URLs targeting the mocked services
    orderService = new OrderService("http://localhost:8081/processPayment",
"http://localhost:8082/checkStock");
}
```

This test validates that the **OrderService** can successfully process an order when the inventory is available, and the payment is processed successfully.

Lastly, create the Integration test that validates that the **OrderService** can successfully process an order when the inventory is available. Post this, the payment is processed successfully.

```
@Test
public void testOrderProcessingSuccess() {
    // Create a test order
    Order testOrder = createTestOrder();

    // Place the order through the OrderService
    OrderResponse response = orderService.placeOrder(testOrder);

    // Assert that the order was processed successfully
    assertEquals("success", response.getStatus(), "Order should be processed successfully");
}
```

Also, make sure to check the Order Service's ability to handle scenarios where an order cannot be processed because the requested items are not available in inventory.

Adjust the Inventory Service's WireMock stub.

```
@Test
public void testOrderProcessingFailureDueInventoryUnavailability() {
    // Adjust the WireMock stub for the Inventory Service to simulate unavailability
    mockInventoryService.stubFor(get(urlEqualTo("/inventory/check"))
        .willReturn(aResponse()
            .withHeader("Content-Type", "application/json")
            .withStatus(200)
            .withBody("{\"available\": false}")));

    // Create a test order
    Order testOrder = createTestOrder();

    // Try to place the order through the OrderService
    OrderResponse response = orderService.placeOrder(testOrder);

    // Assert that the order processing failed due to inventory unavailability
    assertEquals("failure", response.getStatus(), "Order processing should fail due to inventory
unavailability");
}
```

Conclusion

Embracing shift-left testing methodologies, particularly through the strategic integration of service virtualization and contract testing, marks a pivotal advancement in ensuring the quality and reliability of microservices architecture.

By addressing the inherent challenges head-on, organizations can significantly enhance their testing efficiency, accelerate time-to-market, and ultimately deliver superior user experiences. This holistic approach not only mitigates risks associated with microservices but also fosters a culture of continuous improvement and innovation in software development practices.

References

Waterfall model in the nutshell, FourWeekMBA
John Mathon (2015, May 28) MicroServices Martin Fowler, Netflix, Componentized Composable SOA