Testing in a Microservices Architecture

> IE University - BCSAI - SDD - 2025

1. Microservices testing: core challenges

- Network unreliability: latency, timeouts, partial failures
- Data consistency: database-per-service, eventual consistency
- Service dependencies: many teams and many dependencies
- Deployment/versioning independence: API compatibility risks
- Test environment complexity → expensive to replicate

1.1 Network unreliability

- Tests must avoid brittle network dependencies
- Use mocks, service virtualization, and deterministic stubs for unit tests
- Reserve a small, well-maintained set of integration/E2E tests for real-network checks

1.2 Data consistency

- Expect eventual consistency across service boundaries
- Design tests that assert eventual state (polling with timeouts, or callbacks)
- Use integration tests to validate compensations and saga behaviors

1.3 Service dependencies

- Use contract tests to decouple teams and catch breaking changes early
- Virtualize expensive or unstable dependencies during development

1.4 Deployment independence

- Version contracts and run provider verification in CI before deploy
- Use feature flags to decouple rollout from schema changes

1.5 Test environment complexity

- Use layered testing strategy to avoid full-system spins for every change
- Provide reproducible, shared stubs/fixtures for integration tests

2. The Microservices Testing Pyramid

- Unit tests (60–70%): fast, isolated logic tests
- Integration tests (20–25%): infra components, DB, messaging
- Contract tests (10–15%): consumer/provider API expectations
- End-to-end tests (5–10%): critical user journeys only

3. Contract testing & consumer-driven contracts (CDC)

- A contract defines the expected interaction between a consumer (client) and a provider (service).
- It specifies the request format the consumer will send and the response format the provider will return.
- Contracts are typically expressed as executable specifications that can be tested independently by both parties.

CDC workflow (high level)

1. Consumer writes pact/contract tests → produces artifact (JSON)
2. Contract published/shared (repo or broker)
3. Provider verifies contracts in CI against running provider
4. Green = safe independent deployments

3.1 Consumer-side example (Pact)

```
from pact import Consumer, Provider
import pytest
pact = Consumer('OrderService').has_pact_with(Provider('InventoryService'))
def test_get_product_availability():
    expected = {
        'product_id': '12345',
        'available': True,
        'quantity': 100
    }

    (pact
     .given('product 12345 exists with stock')
     .upon_receiving('a request for product availability')
     .with_request('GET', '/api/inventory/12345')
     .will_respond_with(200, body=expected))

    with pact:
        from order_service import InventoryClient
        client = InventoryClient(pact.uri)
        result = client.check_availability('12345')

        assert result['available'] is True
        assert result['quantity'] == 100
```

3.2 Provider verification example (Pact)

```
from pact import Verifier
import pytest
def test_provider_honors_consumer_contracts():
    verifier = Verifier(
        provider='InventoryService',
        provider_base_url='http://localhost:8080'
    )

    success = verifier.verify_pacts(
        './pacts/OrderService-InventoryService.json',
        provider_states_setup_url='http://localhost:8080/pact/provider-states'
    )

    assert success
```

3.3 Benefits of contract testing

- Fast feedback and focused failures
- Consumer-driven specs reduce over-constraining providers
- Avoids costly full-system E2E runs for interface regressions

4. Testing synchronous service interactions

- Unit tests: mock HTTP clients (no network)
- Integration tests: use real HTTP client with endpoint mocking (requests_mock)
- Test timeouts, retries, and error classification (4xx vs 5xx)

4.1 Unit testing with mocked HTTP clients

- Use unittest.mock.patch to replace HTTP client calls
- Assert requests made (URL, headers, params) and handling logic
- Keep unit tests fast and deterministic

```python
import pytest
from unittest.mock import Mock, patch
from order_service import OrderProcessor
class TestOrderProcessor:
    @patch('order_service.http_client.get')
    def test_processes_order_when_inventory_available(self, mock_get):
        mock_get.return_value = Mock(
            status_code=200,
            json=lambda: {'available': True, 'quantity': 50}
        )

        processor = OrderProcessor()
        order = {'product_id': '12345', 'quantity': 10}

        result = processor.process_order(order)

        assert result['status'] == 'confirmed'
        mock_get.assert_called_once_with(
            'http://inventory-service/api/inventory/12345'
        )
```

4.2 Integration testing: timeout scenario

- Use `requests` + `requests_mock` to simulate a timeout
- Verify service returns a graceful 'pending' or retryable response

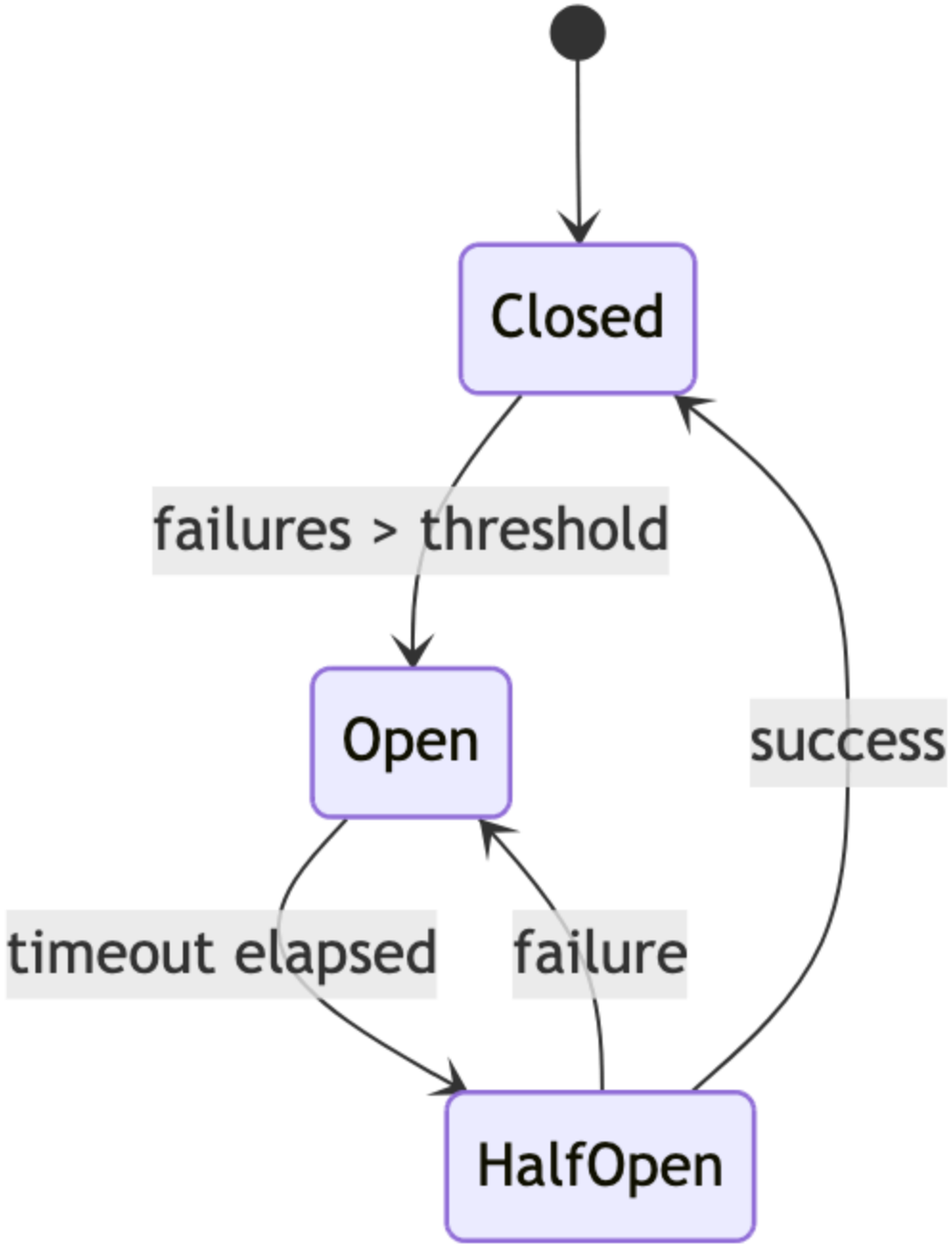
4.3 Integration testing: server error scenario

- Simulate 5xx response and verify retry/backoff behavior or failure mode
- Distinguish between client errors (4xx) and server errors (5xx) in tests

```
    def test_handles_inventory_service_500_error(self, requests_mock):
        requests_mock.get(
            'http://inventory-service/api/inventory/12345',
            status_code=500,
            json={'error': 'Internal server error'}
        )

        processor = OrderProcessor()
        order = {'product_id': '12345', 'quantity': 10}

        result = processor.process_order(order)

        assert result['status'] == 'failed'
        assert 'retry' in result
```

4.4 Resilience: Circuit Breaker

- Circuit breaker prevents cascading failures by failing fast after threshold
- States: Closed → Open → Half-Open → Closed
- Test: N consecutive failures → circuit opens → subsequent calls fail fast


Closed
failures > threshold
Open
success
timeout elapsed
failure
HalfOpen

```python
import pytest
from unittest.mock import Mock, patch
import requests
from order_service import OrderProcessor
class TestResiliencePatterns:
    @patch('order_service.http_client.get')
    def test_circuit_breaker_opens_after_threshold_failures(self, mock_get):
        mock_get.side_effect = requests.exceptions.Timeout

        processor = OrderProcessor()
        order = {'product_id': '12345', 'quantity': 10}

        for i in range(5):
            result = processor.process_order(order)
            assert result['status'] == 'pending'

        initial_call_count = mock_get.call_count

        result = processor.process_order(order)

        assert mock_get.call_count == initial_call_count
        assert result['status'] == 'circuit_open'
```

5. Testing asynchronous communication

- Asynchronous communication through message queues or event streams is common in microservices for decoupling and scalability.
- Testing async patterns requires different approaches than testing synchronous calls.

Message testing approaches

- Unit: mock producer/consumer client APIs; assert calls and serialization
- Integration: run embedded broker and verify publish→consume→effect
- E2E: small, focused flows that exercise multiple services

Error handling & idempotency

- Simulate poison messages and verify consumer moves message to DLQ
- Test idempotent handlers by replaying the same event

5.1 Testing message publishing: verify message schema & headers

```
import pytest
from unittest.mock import Mock, patch
from order_service import OrderService
class TestMessagePublishing:
    @patch('order_service.message_broker.publish')
    def test_publishes_order_created_event(self, mock_publish):
        service = OrderService()
        order_data = {
            'customer_id': 'cust123',
            'items': [{'product_id': 'prod456', 'quantity': 2}],
            'total': 59.99
        }

        order = service.create_order(order_data)

        mock_publish.assert_called_once()

        call_args = mock_publish.call_args

        event = call_args[0][0]

        assert event['event_type'] == 'OrderCreated'
        assert event['order_id'] == order.id
        assert event['customer_id'] == 'cust123'
        assert event['total'] == 59.99

        assert call_args[1]['exchange'] == 'orders'
        assert call_args[1]['routing_key'] == 'order.created'
```

5.2 Testing message consumption: handler logic + side effects

```
import pytest
from inventory_service import InventoryEventHandler
class TestInventoryEventConsumer:
    def test_reduces_stock_on_order_created_event(self):
        handler = InventoryEventHandler()

        event = {
            'event_type': 'OrderCreated',
            'order_id': 'ord123',
            'items': [
                {'product_id': 'prod456', 'quantity': 2},
                {'product_id': 'prod789', 'quantity': 1}
            ]
        }

        handler.handle_order_created(event)

        product_456 = handler.inventory_repo.get('prod456')
        assert product_456.quantity == 98

        product_789 = handler.inventory_repo.get('prod789')
        assert product_789.quantity == 49
```

5.3 Testing error handling: poison messages, retries, DLQs

```
    def test_handles_insufficient_stock_gracefully(self):
        handler = InventoryEventHandler()

        event = {
            'event_type': 'OrderCreated',
            'order_id': 'ord124',
            'items': [
                {'product_id': 'prod456', 'quantity': 200}
            ]
        }

        handler.handle_order_created(event)

        compensation_events = handler.get_published_events()
        assert len(compensation_events) == 1
        assert compensation_events[0]['event_type'] == 'InsufficientStock'
        assert compensation_events[0]['order_id'] == 'ord124'
```

5.4 Testing with embedded brokers: Testcontainers or in-memory brokers

```
import pytest
import testcontainers.rabbitmq
from order_service import OrderService
from inventory_service import InventoryService
@pytest.fixture(scope='module')
def rabbitmq_container():
    with testcontainers.rabbitmq.RabbitMqContainer() as rabbitmq:
        yield rabbitmq
class TestAsyncIntegration:
    def test_order_to_inventory_flow(self, rabbitmq_container):
        order_service = OrderService(
            broker_url=rabbitmq_container.get_connection_url()
        )
        inventory_service = InventoryService(
            broker_url=rabbitmq_container.get_connection_url()
        )

        inventory_service.start_consuming()

        order_data = {
            'customer_id': 'cust123',
            'items': [{'product_id': 'prod456', 'quantity': 2}]
        }
        order = order_service.create_order(order_data)

        inventory_service.wait_for_message_processing(timeout=5)

        product = inventory_service.get_product('prod456')
        assert product.quantity == 98
```

5.5 Testing eventual consistency: assert eventual side effects

```
import pytest
import time
from order_service import OrderService
from notification_service import NotificationService
class TestEventualConsistency:
    def test_customer_notified_after_order_confirmation(self):
        order_service = OrderService()
        notification_service = NotificationService()

        order = order_service.create_order({
            'customer_id': 'cust123',
            'items': [{'product_id': 'prod456', 'quantity': 1}]
        })

        max_wait = 5
        start_time = time.time()

        while time.time() - start_time < max_wait:
            notifications = notification_service.get_notifications('cust123')

            if any(n['order_id'] == order.id for n in notifications):
                break

            time.sleep(0.1)
        else:
            pytest.fail('Notification not received within timeout')

        notification = next(
            n for n in notifications if n['order_id'] == order.id
        )

        assert notification['type'] == 'order_confirmation'
        assert notification['customer_id'] == 'cust123'
```

6. Service virtualization & test doubles

When testing a microservice, you need strategies for handling its dependencies without requiring all services to be running. Service virtualization and test doubles provide this capability.

Types of Test Doubles for Services

- **Mock Services** are programmable test doubles where you define exact expectations about what calls will be made and what responses to return. Mocks are useful for unit tests but become brittle in integration tests.
- **Stub Services** return pre-configured responses without verifying how they're called. Stubs are simpler than mocks and useful when you just need a dependency to respond appropriately.
- **Fake Services** are lightweight implementations that behave like the real service but are simplified. For example, an in-memory database instead of a real database, or a file-based queue instead of RabbitMQ.
- **Service Virtualization** involves using tools that can record real service interactions and replay them, providing realistic test environments without the complexity of running real services.

Example: Using WireMock for HTTP Service Virtualization

```
import pytest
from wiremock import WireMock
from order_service import OrderProcessor
@pytest.fixture
def inventory_service_mock():
    wiremock = WireMock('localhost', 8080)
    wiremock.start()
    yield wiremock
    wiremock.stop()
class TestWithServiceVirtualization:
    def test_processes_order_with_available_inventory(
        self,
        inventory_service_mock
    ):
        inventory_service_mock.stub_for(
            method='GET',
            url='/api/inventory/12345',
            response={
                'status': 200,
                'body': {
                    'product_id': '12345',
                    'available': True,
                    'quantity': 100
                }
            }
        )

        processor = OrderProcessor(
            inventory_url='http://localhost:8080'
        )
        order = {'product_id': '12345', 'quantity': 10}

        result = processor.process_order(order)

        assert result['status'] == 'confirmed'

        requests = inventory_service_mock.get_all_requests()
        assert len(requests) == 1
        assert requests[0]['url'] == '/api/inventory/12345'
```

Contract-Based Stub Generation

```
from pact import MessageProvider
import pytest
@pytest.fixture
def inventory_stub():
    stub = MessageProvider.from_pact_file(
        './pacts/OrderService-InventoryService.json'
    )
    stub.start()
    yield stub
    stub.stop()
def test_with_contract_based_stub(inventory_stub):
    processor = OrderProcessor(
        inventory_url=inventory_stub.url
    )

    order = {'product_id': '12345', 'quantity': 10}
    result = processor.process_order(order)

    assert result['status'] == 'confirmed'
```

Virtualization best practices

- Keep mappings close to consumer tests
- Version and store stubs in source control or a stub registry
- Use virtualization for slow/unavailable/expensive providers

7. Testing distributed transactions

Sagas testing checklist

- Happy path: all steps succeed
- Failure path: simulate failure at each participant and assert compensations
- Concurrency: test duplicate or delayed messages and eventual consistency

7.1 Saga happy path: all participants commit

```
    def test_completes_successfully_when_all_steps_succeed(self):
        saga = OrderSaga()

        saga.inventory_service.reserve_inventory = Mock(
            return_value={'reserved': True, 'reservation_id': 'res123'}
        )
        saga.payment_service.charge_customer = Mock(
            return_value={'transaction_id': 'txn456', 'status': 'completed'}
        )
        saga.inventory_service.confirm_reservation = Mock()
        saga.notification_service.send_confirmation = Mock()

        order = {
            'customer_id': 'cust123',
            'items': [{'product_id': 'prod456', 'quantity': 2}],
            'total': 59.99
        }

        result = saga.execute(order)

        assert result['status'] == 'completed'

        saga.inventory_service.confirm_reservation.assert_called_once()
        saga.notification_service.send_confirmation.assert_called_once()
```

7.2 Saga compensation: simulate failure and verify compensations

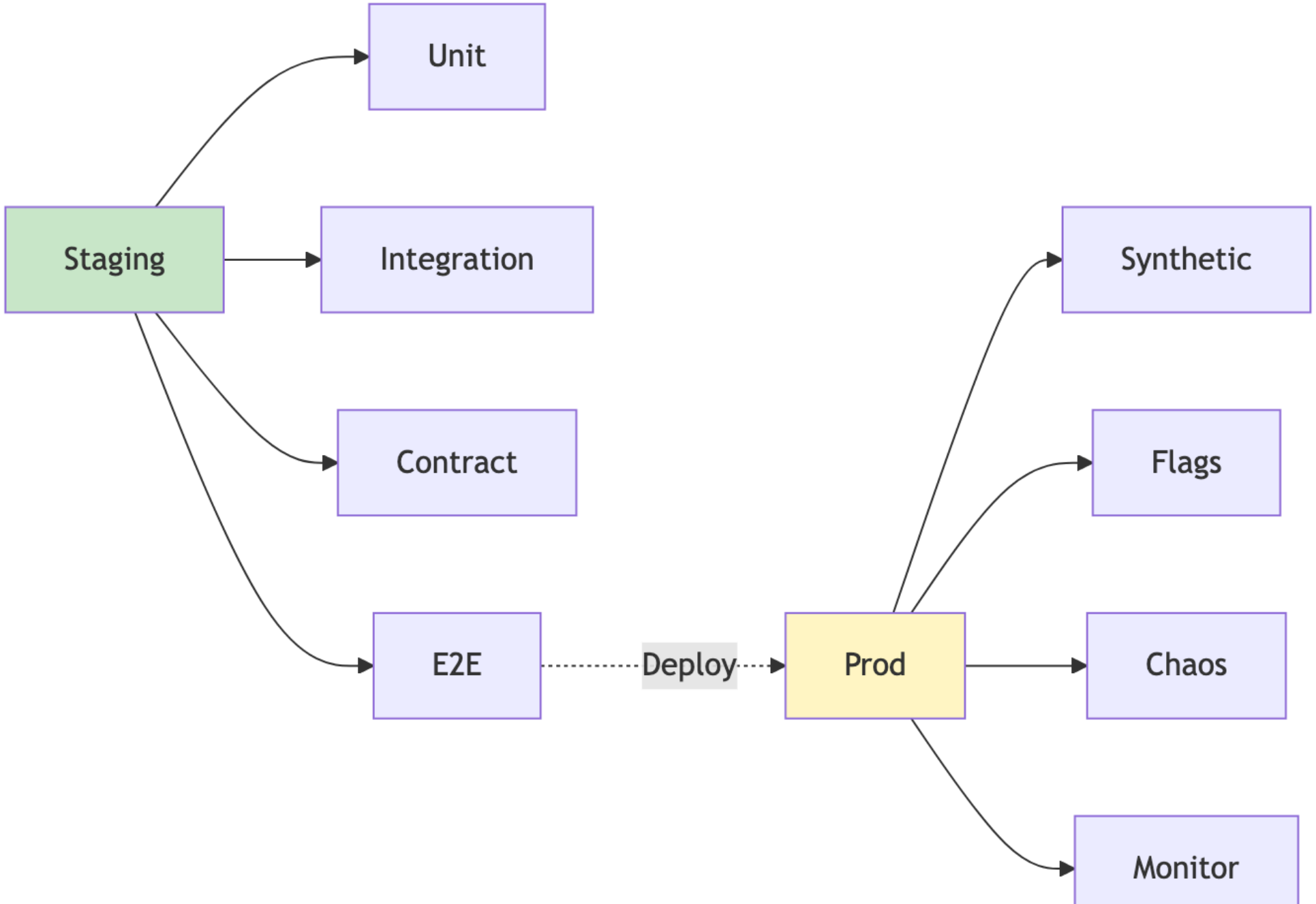
```
import pytest
from order_saga import OrderSaga
from unittest.mock import Mock, patch
class TestOrderSaga:
    def test_compensates_when_payment_fails(self):
        saga = OrderSaga()

        saga.inventory_service.reserve_inventory = Mock(
            return_value={'reserved': True, 'reservation_id': 'res123'}
        )

        saga.payment_service.charge_customer = Mock(
            side_effect=PaymentFailedException('Insufficient funds')
        )

        saga.inventory_service.release_reservation = Mock()

        order = {
            'customer_id': 'cust123',
            'items': [{'product_id': 'prod456', 'quantity': 2}],
            'total': 59.99
        }

        result = saga.execute(order)

        assert result['status'] == 'failed'
        assert result['reason'] == 'payment_failed'

        saga.inventory_service.release_reservation.assert_called_once_with(
            'res123'
        )
```

7.3 Idempotency: ensure retries don't produce duplicates

```
import pytest
from payment_service import PaymentProcessor
class TestIdempotency:
    def test_duplicate_payment_request_does_not_double_charge(self):
        processor = PaymentProcessor()

        payment_request = {
            'idempotency_key': 'unique-key-123',
            'customer_id': 'cust123',
            'amount': 59.99
        }

        result1 = processor.process_payment(payment_request)
        assert result1['status'] == 'completed'
        transaction_id1 = result1['transaction_id']

        result2 = processor.process_payment(payment_request)
        assert result2['status'] == 'completed'

        assert result2['transaction_id'] == transaction_id1

        charges = processor.get_charges_for_customer('cust123')
        assert len(charges) == 1
        assert charges[0]['amount'] == 59.99
```

8. Observability & testing in production

In microservices, comprehensive testing before deployment is necessary but not sufficient. Observability and testing in production become essential practices.


Unit
Staging
Integration
Contract
E2E
Deploy
Prod
Synthetic
Flags
Chaos
Monitor

8.1 Synthetic transactions for continuous E2E monitoring

```
import pytest
import schedule
import time
from monitoring import MetricsCollector
class SyntheticTransactionMonitor:
    def __init__(self, metrics_collector):
        self.metrics = metrics_collector

    def test_create_order_flow(self):
        start_time = time.time()

        try:
            order = self.create_synthetic_order()

            status = self.check_order_status(order['id'])

            if status == 'completed':
                duration = time.time() - start_time
                self.metrics.record_success('order_flow', duration)
            else:
                self.metrics.record_failure('order_flow', 'incomplete')

        except Exception as e:
            self.metrics.record_failure('order_flow', str(e))

    def create_synthetic_order(self):
        return {
            'customer_id': 'synthetic-test-customer',
            'items': [{'product_id': 'test-product', 'quantity': 1}],
            'synthetic': True
        }
def run_synthetic_monitoring():
    monitor = SyntheticTransactionMonitor(MetricsCollector())

    schedule.every(5).minutes.do(monitor.test_create_order_flow)

    while True:
        schedule.run_pending()
        time.sleep(1)
```

8.2 Feature flags for progressive rollout and safe experimentation

```
import pytest
from feature_flags import FeatureFlags
from order_service import OrderProcessor
class TestWithFeatureFlags:
    def test_new_pricing_algorithm_with_flag_enabled(self):
        flags = FeatureFlags()

        flags.enable('new_pricing_algorithm', percentage=10)

        processor = OrderProcessor(feature_flags=flags)

        order = {
            'customer_id': 'cust123',
            'items': [{'product_id': 'prod456', 'quantity': 2}]
        }

        result = processor.calculate_total(order)

        assert result['pricing_version'] == 'v2'
        assert result['total'] < 100
```

8.3 Chaos engineering: targeted experiments (latency, partial failures)

```
import pytest
from chaos_engineering import ChaosMonkey
from order_service import OrderService
class TestResilience:
    def test_system_handles_inventory_service_failure(self):
        chaos = ChaosMonkey()

        chaos.kill_service('inventory-service')

        order_service = OrderService()
        order = {
            'customer_id': 'cust123',
            'items': [{'product_id': 'prod456', 'quantity': 2}]
        }

        result = order_service.create_order(order)

        assert result['status'] == 'pending'
        assert result['reason'] == 'inventory_unavailable'

        chaos.restore_service('inventory-service')
```

8.4 Chaos for latency testing and boundary conditions

```
    def test_system_handles_increased_latency(self):
        chaos = ChaosMonkey()

        chaos.add_latency('payment-service', delay_ms=5000)

        order_service = OrderService()
        order = {
            'customer_id': 'cust123',
            'items': [{'product_id': 'prod456', 'quantity': 2}]
        }

        import time
        start = time.time()
        result = order_service.create_order(order)
        duration = time.time() - start

        assert duration < 3.0
        assert result['status'] == 'timeout'

        chaos.remove_latency('payment-service')
```

Production testing practices

- Keep synthetic tests minimal and low-risk
- Combine feature flags with canarying and monitoring
- Automate rollback and alerting for experiment failures