Temporal Training Session 11: Testing Temporal Workflows and Activities

Table of Contents

Welcome and Recap

Welcome to Day 11 of our Temporal training. Over the past ten days, we've explored the fundamentals of Temporal, from basic workflows and activities to advanced patterns like child workflows, versioning, and dynamic registration. Today, we shift our focus to a critical aspect of production-ready Temporal applications: **testing**.

Testing Temporal applications presents unique challenges due to their distributed nature, stateful behavior, and the need for determinism. We'll explore comprehensive testing strategies that ensure your workflows and activities are reliable, maintainable, and production-ready.

Why Testing is Critical in Temporal

Testing in Temporal is not just about ensuring code correctness—it's about guaranteeing **durable execution reliability**. Unlike traditional applications where a simple unit test might suffice, Temporal workflows have several characteristics that make testing particularly important:

Unique Testing Challenges in Temporal

1. **Determinism Requirements:** Workflows must be deterministic, meaning the same input must always produce the same output. Testing helps catch non-deterministic code that could break replay.

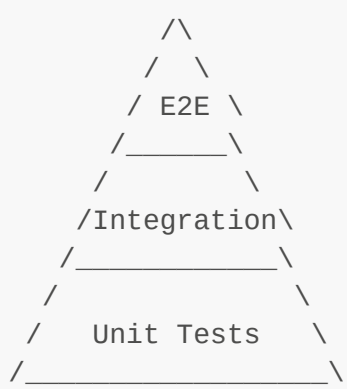
2. **Stateful Nature:** Workflows maintain state across executions, making it essential to test state transitions and persistence.

3. **Distributed Execution:** Activities run on different workers, requiring testing of distributed coordination and failure scenarios.

4. **Long-Running Processes:** Workflows can run for days or months, making traditional testing approaches insufficient.

5. **Event Sourcing:** The event history must be consistent and replayable, requiring tests that validate the complete execution path.

Testing Pyramid for Temporal Applications



- **Unit Tests:** Test individual workflows and activities in isolation
- **Integration Tests:** Test workflows with a real Temporal server
- **End-to-End Tests:** Test complete business processes

Testing Strategies Overview

Temporal provides several testing approaches, each serving different purposes:

Testing Type	Purpose	Tools	When to Use
Unit Tests	Test workflow/activity logic in isolation	`temporalio.testing.WorkflowEnvironment`	During development, CI/CD
Integration Tests	Test with real Temporal server	`temporalio.testing.WorkflowEnvironment`	Pre-deployment validation
End-to-End Tests	Test complete business processes	Full Temporal cluster	Production readiness

Key Testing Principles

1. **Test Determinism:** Ensure workflows produce consistent results
2. **Test Failure Scenarios:** Validate error handling and recovery
3. **Test State Transitions:** Verify workflow state changes
4. **Test Timeouts and Retries:** Ensure robust error handling
5. **Test External Dependencies:** Mock external services appropriately

---

Unit Testing Workflows

Unit testing workflows involves testing the workflow logic in isolation, without requiring a full Temporal server. This approach is fast, reliable, and suitable for CI/CD pipelines.

Setting Up the Test Environment

```
# test_workflows.py
import pytest
from temporalio.testing import WorkflowEnvironment
from temporalio.client import Client
from temporalio.worker import Worker
from your_workflows import GreetingWorkflow
from your_activities import compose_greeting

@pytest.fixture
async def temporal_client():
    """Create a test Temporal client."""
    async with WorkflowEnvironment.start_local() as env:
        client = await env.client()
        yield client
        await env.shutdown()
```

Basic Workflow Testing

```
async def test_greeting_workflow_success(temporal_client):
    """Test that the greeting workflow returns the expected result."""

    # Start the workflow
    result = await temporal_client.execute_workflow(
        GreetingWorkflow.run,
        "Alice",
        id="test-greeting-workflow",
        task_queue="test-task-queue"
    )

    # Assert the expected result
    assert result == "Hello, Alice!"
```

Testing Workflow State and Logic

```
async def test_workflow_with_sleep_and_activity(temporal_client):
    """Test workflow that includes sleep and activity execution."""

    # Start the workflow
    result = await temporal_client.execute_workflow(
        DelayedGreetingWorkflow.run,
```

```
        "Bob",
        id="test-delayed-greeting",
        task_queue="test-task-queue"
    )

    # Verify the workflow completed successfully
    assert result == "Hello, Bob! (delayed)"

    # You can also verify workflow execution details
    handle = temporal_client.get_workflow_handle("test-delayed-greeting")
    desc = await handle.describe()
    assert desc.status.name == "COMPLETED"
```

Testing Error Scenarios

```
async def test_workflow_with_activity_failure(temporal_client):
    """Test workflow behavior when activities fail."""

    # Mock the activity to raise an exception
    with patch('your_activities.compose_greeting') as mock_activity:
        mock_activity.side_effect = Exception("API Error")

        # Start the workflow
        with pytest.raises(Exception):
            await temporal_client.execute_workflow(
                GreetingWorkflow.run,
                "Charlie",
                id="test-failure-workflow",
                task_queue="test-task-queue"
            )
```

Unit Testing Activities

Activities can be tested more like traditional functions, but with considerations for their role in the Temporal ecosystem.

Testing Activity Logic

```
# test_activities.py
import pytest
from your_activities import compose_greeting, process_payment

def test_compose_greeting_activity():
    """Test the compose_greeting activity logic."""
    result = compose_greeting("David")
    assert result == "Hello, David!"

def test_process_payment_activity_success():
```

```
    """Test successful payment processing."""
    payment_data = {
        "amount": 100.00,
        "currency": "USD",
        "card_number": "4111111111111111"
    }

    result = process_payment(payment_data)
    assert result["status"] == "success"
    assert result["transaction_id"] is not None
```

Testing Activity with External Dependencies

```
from unittest.mock import patch, Mock

def test_process_payment_with_mock_payment_gateway():
    """Test payment processing with mocked external API."""

    # Mock the payment gateway response
    mock_response = Mock()
    mock_response.status_code = 200
    mock_response.json.return_value = {
        "transaction_id": "txn_12345",
        "status": "succeeded"
    }

    with patch('requests.post', return_value=mock_response):
        payment_data = {
            "amount": 50.00,
            "currency": "USD",
            "card_number": "4111111111111111"
        }

        result = process_payment(payment_data)
        assert result["status"] == "success"
        assert result["transaction_id"] == "txn_12345"
```

Testing Activity Retry Logic

```
def test_activity_retry_on_failure():
    """Test that activities retry appropriately on failure."""

    call_count = 0

    def failing_activity():
        nonlocal call_count
        call_count += 1
        if call_count < 3:
            raise Exception("Temporary failure")
```

```
        return "Success after retries"

    # Simulate retry logic
    max_retries = 3
    for attempt in range(max_retries):
        try:
            result = failing_activity()
            break
        except Exception:
            if attempt == max_retries - 1:
                raise

    assert result == "Success after retries"
    assert call_count == 3
```

Mocking External Dependencies

External dependencies in Temporal applications (APIs, databases, etc.) should be mocked to ensure tests are fast, reliable, and don't depend on external services.

Mocking HTTP APIs

```
# test_external_apis.py
import pytest
from unittest.mock import patch, Mock
from your_activities import fetch_user_data, send_notification

def test_fetch_user_data_with_mock_api():
    """Test user data fetching with mocked API."""

    # Mock API response
    mock_response = Mock()
    mock_response.status_code = 200
    mock_response.json.return_value = {
        "id": 123,
        "name": "John Doe",
        "email": "john@example.com"
    }

    with patch('requests.get', return_value=mock_response):
        user_data = fetch_user_data(123)

        assert user_data["id"] == 123
        assert user_data["name"] == "John Doe"
        assert user_data["email"] == "john@example.com"

def test_fetch_user_data_api_failure():
    """Test handling of API failures."""

    # Mock API failure
```

```
    mock_response = Mock()
    mock_response.status_code = 404
    mock_response.raise_for_status.side_effect = Exception("User not
found")

    with patch('requests.get', return_value=mock_response):
        with pytest.raises(Exception, match="User not found"):
            fetch_user_data(999)
```

Mocking Database Operations

```
# test_database_operations.py
import pytest
from unittest.mock import patch, Mock
from your_activities import save_order, get_order_status

def test_save_order_with_mock_database():
    """Test order saving with mocked database."""

    # Mock database connection and cursor
    mock_cursor = Mock()
    mock_cursor.fetchone.return_value = (1,)  # Return order ID

    mock_connection = Mock()
    mock_connection.cursor.return_value.__enter__.return_value =
mock_cursor

    with patch('your_activities.get_database_connection',
return_value=mock_connection):
        order_data = {
            "customer_id": 123,
            "items": [{"product_id": 1, "quantity": 2}],
            "total": 100.00
        }

        order_id = save_order(order_data)
        assert order_id == 1

        # Verify the correct SQL was executed
        mock_cursor.execute.assert_called_once()
        call_args = mock_cursor.execute.call_args[0][0]
        assert "INSERT INTO orders" in call_args
```

Mocking Time-Dependent Operations

```
# test_time_operations.py
import pytest
from unittest.mock import patch
from datetime import datetime, timedelta
```

```
from your_activities import check_scheduled_task

def test_check_scheduled_task_with_mock_time():
    """Test time-dependent logic with mocked time."""

    # Mock current time
    mock_now = datetime(2024, 1, 15, 10, 30, 0)

    with patch('datetime.datetime') as mock_datetime:
        mock_datetime.now.return_value = mock_now

        # Test task that should be executed
        scheduled_time = datetime(2024, 1, 15, 10, 0, 0)  # 30 minutes ago
        result = check_scheduled_task(scheduled_time)
        assert result is True

        # Test task that should not be executed yet
        future_time = datetime(2024, 1, 15, 11, 0, 0)  # 30 minutes in
future
        result = check_scheduled_task(future_time)
        assert result is False
```

Integration Testing with Temporal Server

Integration tests use a real Temporal server to test the complete workflow execution, including server interactions, task queuing, and worker coordination.

Setting Up Integration Tests

```
# test_integration.py
import pytest
import asyncio
from temporalio.testing import WorkflowEnvironment
from temporalio.client import Client
from temporalio.worker import Worker
from your_workflows import OrderProcessingWorkflow
from your_activities import validate_order, process_payment,
send_confirmation

@pytest.fixture
async def temporal_environment():
    """Set up a complete Temporal environment for integration testing."""
    async with WorkflowEnvironment.start_local() as env:
        # Create client
        client = await env.client()

        # Start worker
        worker = Worker(
            client,
            task_queue="integration-test-queue",
```

```
            workflows=[OrderProcessingWorkflow],
            activities=[validate_order, process_payment, send_confirmation]
        )

        # Start worker in background
        worker_task = asyncio.create_task(worker.run())

        yield client, worker

        # Cleanup
        worker_task.cancel()
        try:
            await worker_task
        except asyncio.CancelledError:
            pass
        await env.shutdown()
```

Testing Complete Workflow Execution

```
async def test_order_processing_workflow_integration(temporal_environment):
    """Test complete order processing workflow with real Temporal
server."""

    client, worker = temporal_environment

    # Prepare test data
    order_data = {
        "customer_id": 123,
        "items": [
            {"product_id": 1, "quantity": 2, "price": 25.00},
            {"product_id": 2, "quantity": 1, "price": 50.00}
        ],
        "payment_method": "credit_card",
        "card_number": "4111111111111111"
    }

    # Execute the workflow
    result = await client.execute_workflow(
        OrderProcessingWorkflow.run,
        order_data,
        id="test-order-processing",
        task_queue="integration-test-queue"
    )

    # Verify the result
    assert result["status"] == "completed"
    assert result["order_id"] is not None
    assert result["transaction_id"] is not None
    assert result["confirmation_sent"] is True

    # Verify workflow execution details
```

```
    handle = client.get_workflow_handle("test-order-processing")
    desc = await handle.describe()
    assert desc.status.name == "COMPLETED"
```

Testing Workflow with Signals and Queries

```
async def test_workflow_with_signals_and_queries(temporal_environment):
    """Test workflow that responds to signals and queries."""

    client, worker = temporal_environment

    # Start the workflow
    handle = await client.start_workflow(
        ApprovalWorkflow.run,
        {"request_id": "req_123", "amount": 1000.00},
        id="test-approval-workflow",
        task_queue="integration-test-queue"
    )

    # Query the workflow state
    state = await handle.query(ApprovalWorkflow.get_state)
    assert state["status"] == "pending_approval"

    # Send approval signal
    await handle.signal(ApprovalWorkflow.approve, {"approver_id":
"user_456"})

    # Wait for completion
    result = await handle.result()
    assert result["status"] == "approved"
    assert result["approver_id"] == "user_456"
```

Testing Error Scenarios and Recovery

```
async def test_workflow_error_recovery_integration(temporal_environment):
    """Test workflow recovery from activity failures."""

    client, worker = temporal_environment

    # Start workflow with data that will cause an activity to fail
    order_data = {
        "customer_id": 123,
        "items": [{"product_id": 999, "quantity": 1, "price": 100.00}],  #
Invalid product
        "payment_method": "credit_card",
        "card_number": "4111111111111111"
    }

    # Execute workflow and expect it to handle the error gracefully
```

```
    result = await client.execute_workflow(
        OrderProcessingWorkflow.run,
        order_data,
        id="test-error-recovery",
        task_queue="integration-test-queue"
    )

    # Verify error handling
    assert result["status"] == "failed"
    assert "Invalid product" in result["error_message"]
    assert result["compensation_applied"] is True
```

Testing Best Practices

1. Test Structure and Organization

```
# Organize tests by functionality
class TestOrderWorkflows:
    """Test suite for order-related workflows."""

    async def test_order_creation_success(self, temporal_client):
        """Test successful order creation."""
        pass

    async def test_order_creation_validation_failure(self,
temporal_client):
        """Test order creation with invalid data."""
        pass

    async def test_order_cancellation(self, temporal_client):
        """Test order cancellation workflow."""
        pass

class TestPaymentActivities:
    """Test suite for payment-related activities."""

    def test_payment_processing_success(self):
        """Test successful payment processing."""
        pass

    def test_payment_processing_insufficient_funds(self):
        """Test payment processing with insufficient funds."""
        pass
```

2. Test Data Management

```
# Use fixtures for common test data
@pytest.fixture
```

```
def sample_order_data():
    """Provide sample order data for tests."""
    return {
        "customer_id": 123,
        "items": [
            {"product_id": 1, "quantity": 2, "price": 25.00},
            {"product_id": 2, "quantity": 1, "price": 50.00}
        ],
        "payment_method": "credit_card",
        "card_number": "4111111111111111"
    }

@pytest.fixture
def sample_user_data():
    """Provide sample user data for tests."""
    return {
        "id": 123,
        "name": "John Doe",
        "email": "john@example.com",
        "credit_limit": 1000.00
    }
```

3. Determinism Testing

```
async def test_workflow_determinism(temporal_client):
    """Test that workflow produces consistent results."""

    # Run the same workflow multiple times
    results = []
    for i in range(3):
        result = await temporal_client.execute_workflow(
            DeterministicWorkflow.run,
            {"input": "test"},
            id=f"determinism-test-{i}",
            task_queue="test-task-queue"
        )
        results.append(result)

    # All results should be identical
    assert len(set(results)) == 1
    assert results[0] == results[1] == results[2]
```

4. Performance Testing

```
import time

async def test_workflow_performance(temporal_client):
    """Test workflow execution performance."""
```

```
    start_time = time.time()

    result = await temporal_client.execute_workflow(
        PerformanceTestWorkflow.run,
        {"iterations": 1000},
        id="performance-test",
        task_queue="test-task-queue"
    )

    execution_time = time.time() - start_time

    # Assert performance requirements
    assert execution_time < 5.0  # Should complete within 5 seconds
    assert result["processed_items"] == 1000
```

Practical: Comprehensive Testing Example

Let's create a comprehensive testing example for an e-commerce order processing system.

The System Under Test

```
# workflows.py
from datetime import timedelta
from temporalio import workflow
from .activities import validate_order, process_payment, update_inventory,
send_confirmation

@workflow.defn
class OrderProcessingWorkflow:
    @workflow.run
    async def run(self, order_data: dict) -> dict:
        """Process a complete order workflow."""

        # Step 1: Validate order
        validation_result = await workflow.execute_activity(
            validate_order,
            order_data,
            start_to_close_timeout=timedelta(seconds=30)
        )

        if not validation_result["valid"]:
            return {
                "status": "failed",
                "error_message": validation_result["error"],
                "order_id": None
            }

        # Step 2: Process payment
        payment_result = await workflow.execute_activity(
            process_payment,
```

```
            {
                "amount": validation_result["total_amount"],
                "payment_method": order_data["payment_method"],
                "card_number": order_data["card_number"]
            },
            start_to_close_timeout=timedelta(seconds=60),
            retry_policy=workflow.RetryPolicy(
                initial_interval=timedelta(seconds=1),
                maximum_interval=timedelta(seconds=10),
                maximum_attempts=3
            )
        )

        if payment_result["status"] != "success":
            return {
                "status": "failed",
                "error_message": "Payment processing failed",
                "order_id": None
            }

        # Step 3: Update inventory
        inventory_result = await workflow.execute_activity(
            update_inventory,
            order_data["items"],
            start_to_close_timeout=timedelta(seconds=30)
        )

        if not inventory_result["success"]:
            # Compensate for successful payment
            await workflow.execute_activity(
                process_payment,
                {
                    "amount": -validation_result["total_amount"],
                    "payment_method": order_data["payment_method"],
                    "card_number": order_data["card_number"],
                    "refund": True
                },
                start_to_close_timeout=timedelta(seconds=60)
            )

            return {
                "status": "failed",
                "error_message": "Inventory update failed",
                "order_id": None,
                "compensation_applied": True
            }

        # Step 4: Send confirmation
        await workflow.execute_activity(
            send_confirmation,
            {
                "customer_id": order_data["customer_id"],
                "order_id": inventory_result["order_id"],
                "total_amount": validation_result["total_amount"]
```

```
            },
            start_to_close_timeout=timedelta(seconds=30)
        )

        return {
            "status": "completed",
            "order_id": inventory_result["order_id"],
            "transaction_id": payment_result["transaction_id"],
            "total_amount": validation_result["total_amount"]
        }
```

Comprehensive Test Suite

```
# test_order_processing_comprehensive.py
import pytest
from unittest.mock import patch, Mock
from temporalio.testing import WorkflowEnvironment
from temporalio.client import Client
from temporalio.worker import Worker
from workflows import OrderProcessingWorkflow
from activities import validate_order, process_payment, update_inventory,
send_confirmation

class TestOrderProcessingComprehensive:
    """Comprehensive test suite for order processing workflow."""

    @pytest.fixture
    async def temporal_environment(self):
        """Set up Temporal environment for testing."""
        async with WorkflowEnvironment.start_local() as env:
            client = await env.client()

            worker = Worker(
                client,
                task_queue="comprehensive-test-queue",
                workflows=[OrderProcessingWorkflow],
                activities=[validate_order, process_payment,
update_inventory, send_confirmation]
            )

            worker_task = asyncio.create_task(worker.run())

            yield client, worker

            worker_task.cancel()
            try:
                await worker_task
            except asyncio.CancelledError:
                pass
            await env.shutdown()
```

```python
    @pytest.fixture
    def valid_order_data(self):
        """Provide valid order data for tests."""
        return {
            "customer_id": 123,
            "items": [
                {"product_id": 1, "quantity": 2, "price": 25.00},
                {"product_id": 2, "quantity": 1, "price": 50.00}
            ],
            "payment_method": "credit_card",
            "card_number": "4111111111111111"
        }

    async def test_successful_order_processing(self, temporal_environment, valid_order_data):
        """Test complete successful order processing."""

        client, worker = temporal_environment

        # Mock all activities
        with patch('activities.validate_order') as mock_validate, \
             patch('activities.process_payment') as mock_payment, \
             patch('activities.update_inventory') as mock_inventory, \
             patch('activities.send_confirmation') as mock_confirmation:

            # Configure mocks
            mock_validate.return_value = {
                "valid": True,
                "total_amount": 100.00
            }

            mock_payment.return_value = {
                "status": "success",
                "transaction_id": "txn_12345"
            }

            mock_inventory.return_value = {
                "success": True,
                "order_id": "order_67890"
            }

            mock_confirmation.return_value = {"sent": True}

            # Execute workflow
            result = await client.execute_workflow(
                OrderProcessingWorkflow.run,
                valid_order_data,
                id="test-successful-order",
                task_queue="comprehensive-test-queue"
            )

            # Verify result
            assert result["status"] == "completed"
            assert result["order_id"] == "order_67890"
```

```
            assert result["transaction_id"] == "txn_12345"
            assert result["total_amount"] == 100.00

            # Verify activity calls
            mock_validate.assert_called_once_with(valid_order_data)
            mock_payment.assert_called_once()
            mock_inventory.assert_called_once()
            mock_confirmation.assert_called_once()

    async def test_order_validation_failure(self, temporal_environment,
valid_order_data):
        """Test workflow behavior when order validation fails."""

        client, worker = temporal_environment

        with patch('activities.validate_order') as mock_validate:
            mock_validate.return_value = {
                "valid": False,
                "error": "Invalid product ID"
            }

            result = await client.execute_workflow(
                OrderProcessingWorkflow.run,
                valid_order_data,
                id="test-validation-failure",
                task_queue="comprehensive-test-queue"
            )

            assert result["status"] == "failed"
            assert result["error_message"] == "Invalid product ID"
            assert result["order_id"] is None

    async def test_payment_processing_failure(self, temporal_environment,
valid_order_data):
        """Test workflow behavior when payment processing fails."""

        client, worker = temporal_environment

        with patch('activities.validate_order') as mock_validate, \
             patch('activities.process_payment') as mock_payment:

            mock_validate.return_value = {
                "valid": True,
                "total_amount": 100.00
            }

            mock_payment.return_value = {
                "status": "failed",
                "error": "Insufficient funds"
            }

            result = await client.execute_workflow(
                OrderProcessingWorkflow.run,
                valid_order_data,
```

```
                id="test-payment-failure",
                task_queue="comprehensive-test-queue"
            )

            assert result["status"] == "failed"
            assert result["error_message"] == "Payment processing failed"
            assert result["order_id"] is None

    async def test_inventory_update_failure_with_compensation(self,
temporal_environment, valid_order_data):
        """Test workflow compensation when inventory update fails."""

        client, worker = temporal_environment

        with patch('activities.validate_order') as mock_validate, \
             patch('activities.process_payment') as mock_payment, \
             patch('activities.update_inventory') as mock_inventory:

            mock_validate.return_value = {
                "valid": True,
                "total_amount": 100.00
            }

            mock_payment.return_value = {
                "status": "success",
                "transaction_id": "txn_12345"
            }

            mock_inventory.return_value = {
                "success": False,
                "error": "Product out of stock"
            }

            result = await client.execute_workflow(
                OrderProcessingWorkflow.run,
                valid_order_data,
                id="test-inventory-failure",
                task_queue="comprehensive-test-queue"
            )

            assert result["status"] == "failed"
            assert result["error_message"] == "Inventory update failed"
            assert result["order_id"] is None
            assert result["compensation_applied"] is True

            # Verify that payment refund was attempted
            assert mock_payment.call_count == 2  # Original payment +
refund

    async def test_workflow_determinism(self, temporal_environment,
valid_order_data):
        """Test that workflow produces consistent results across multiple
runs."""
```

```
        client, worker = temporal_environment

        results = []
        for i in range(3):
            with patch('activities.validate_order') as mock_validate, \
                 patch('activities.process_payment') as mock_payment, \
                 patch('activities.update_inventory') as mock_inventory, \
                 patch('activities.send_confirmation') as
mock_confirmation:

                mock_validate.return_value = {
                    "valid": True,
                    "total_amount": 100.00
                }

                mock_payment.return_value = {
                    "status": "success",
                    "transaction_id": f"txn_{i}"
                }

                mock_inventory.return_value = {
                    "success": True,
                    "order_id": f"order_{i}"
                }

                mock_confirmation.return_value = {"sent": True}

                result = await client.execute_workflow(
                    OrderProcessingWorkflow.run,
                    valid_order_data,
                    id=f"determinism-test-{i}",
                    task_queue="comprehensive-test-queue"
                )

                results.append(result["status"])

        # All results should be identical
        assert len(set(results)) == 1
        assert results[0] == "completed"

    async def test_workflow_timeout_handling(self, temporal_environment,
valid_order_data):
        """Test workflow behavior with activity timeouts."""

        client, worker = temporal_environment

        with patch('activities.validate_order') as mock_validate:
            # Simulate slow validation
            async def slow_validation(data):
                await asyncio.sleep(35)  # Exceeds 30-second timeout
                return {"valid": True, "total_amount": 100.00}

            mock_validate.side_effect = slow_validation
```

```
            with pytest.raises(Exception):  # Should timeout
                await client.execute_workflow(
                    OrderProcessingWorkflow.run,
                    valid_order_data,
                    id="test-timeout",
                    task_queue="comprehensive-test-queue"
                )
```

This comprehensive testing approach ensures that your Temporal workflows and activities are thoroughly tested for correctness, reliability, and production readiness. The combination of unit tests, integration tests, and comprehensive scenarios provides confidence that your workflows will behave correctly in production environments.