Temporal Training Session 10: Advanced Workflow Patterns and Best Practices

Table of Contents

Introduction and Recap

Welcome to Day 10 of our Temporal training. Over the past nine days, we've built a comprehensive understanding of Temporal's core concepts, from basic workflows and activities to advanced patterns like child workflows and versioning. Today, we explore the most advanced patterns and best practices that will help you build production-grade Temporal applications.

Learning Objectives:

- Understand dynamic activity and workflow registration
- Learn to use memo and search attributes for enhanced visibility
- Implement custom data converters for complex payload serialization
- Master advanced workflow patterns for complex business scenarios
- Apply performance optimization and security best practices

Dynamic Activity and Workflow Registration

Why Dynamic Registration?

Traditional workflow registration happens at worker startup, but there are scenarios where you need to register activities or workflows dynamically:

1. **Plugin Systems:** Loading activities from external modules
2. **Feature Flags:** Enabling/disabling functionality at runtime
3. **Multi-tenant Systems:** Different tenants need different activities
4. **Testing:** Mocking activities for testing scenarios

Dynamic Activity Registration

```
from temporalio import activity, workflow
from typing import Dict, Any, Callable
import importlib

class DynamicActivityRegistry:
    def __init__(self):
        self.activities: Dict[str, Callable] = {}

    def register_activity(self, name: str, func: Callable):
        """Register an activity dynamically."""
        self.activities[name] = func

    def get_activity(self, name: str) -> Callable:
        """Get a registered activity by name."""
        return self.activities.get(name)

    def load_activities_from_module(self, module_name: str):
        """Load activities from a Python module."""
        try:
            module = importlib.import_module(module_name)

            # Look for functions decorated with @activity.defn
            for attr_name in dir(module):
                attr = getattr(module, attr_name)
                if hasattr(attr, '__temporal_activity_defn'):
                    self.register_activity(attr_name, attr)

        except ImportError as e:
            workflow.logger.error(f"Failed to load module {module_name}:
{e}")

# Global registry instance
activity_registry = DynamicActivityRegistry()

@workflow.defn
class DynamicWorkflow:
    @workflow.run
    async def run(self, activity_name: str, input_data: dict) -> dict:
        # Get activity from registry
        activity_func = activity_registry.get_activity(activity_name)

        if not activity_func:
            raise ValueError(f"Activity {activity_name} not found")

        # Execute the activity
        result = await workflow.execute_activity(
            activity_func,
            input_data,
            start_to_close_timeout=timedelta(seconds=30)
        )
```

```
        return result
```

Dynamic Workflow Registration

```
class DynamicWorkflowRegistry:
    def __init__(self):
        self.workflows: Dict[str, type] = {}

    def register_workflow(self, name: str, workflow_class: type):
        """Register a workflow class dynamically."""
        self.workflows[name] = workflow_class

    def get_workflow(self, name: str) -> type:
        """Get a registered workflow by name."""
        return self.workflows.get(name)

    def load_workflows_from_module(self, module_name: str):
        """Load workflows from a Python module."""
        try:
            module = importlib.import_module(module_name)

            # Look for classes decorated with @workflow.defn
            for attr_name in dir(module):
                attr = getattr(module, attr_name)
                if hasattr(attr, '__temporal_workflow_defn'):
                    self.register_workflow(attr_name, attr)

        except ImportError as e:
            workflow.logger.error(f"Failed to load module {module_name}:
{e}")

# Global registry instance
workflow_registry = DynamicWorkflowRegistry()

# Example usage in a worker
async def run_dynamic_worker():
    client = await Client.connect("localhost:7233")

    # Load activities and workflows dynamically
    activity_registry.load_activities_from_module("my_activities")
    workflow_registry.load_workflows_from_module("my_workflows")

    # Create worker with dynamically loaded components
    worker = Worker(
        client,
        task_queue="dynamic-queue",
        workflows=list(workflow_registry.workflows.values()),
        activities=list(activity_registry.activities.values())
    )
```

```
    await worker.run()
```

Plugin System Example

```
# Plugin interface
class TemporalPlugin:
    def get_activities(self) -> Dict[str, Callable]:
        """Return activities to register."""
        raise NotImplementedError

    def get_workflows(self) -> Dict[str, type]:
        """Return workflows to register."""
        raise NotImplementedError

# Example plugin
class PaymentPlugin(TemporalPlugin):
    def get_activities(self) -> Dict[str, Callable]:
        return {
            "process_payment": self.process_payment,
            "refund_payment": self.refund_payment
        }

    def get_workflows(self) -> Dict[str, type]:
        return {
            "PaymentWorkflow": PaymentWorkflow
        }

    @activity.defn
    async def process_payment(self, amount: float, currency: str) -> dict:
        # Payment processing logic
        return {"status": "success", "transaction_id": "txn_123"}

    @activity.defn
    async def refund_payment(self, transaction_id: str) -> dict:
        # Refund logic
        return {"status": "refunded", "transaction_id": transaction_id}

# Plugin manager
class PluginManager:
    def __init__(self):
        self.plugins: Dict[str, TemporalPlugin] = {}

    def register_plugin(self, name: str, plugin: TemporalPlugin):
        """Register a plugin."""
        self.plugins[name] = plugin

    def get_all_activities(self) -> Dict[str, Callable]:
        """Get all activities from all plugins."""
        activities = {}
        for plugin_name, plugin in self.plugins.items():
```

```
            plugin_activities = plugin.get_activities()
            for activity_name, activity_func in plugin_activities.items():
                activities[f"{plugin_name}.{activity_name}"] =
activity_func
        return activities

    def get_all_workflows(self) -> Dict[str, type]:
        """Get all workflows from all plugins."""
        workflows = {}
        for plugin_name, plugin in self.plugins.items():
            plugin_workflows = plugin.get_workflows()
            for workflow_name, workflow_class in plugin_workflows.items():
                workflows[f"{plugin_name}.{workflow_name}"] =
workflow_class
        return workflows
```

Using Memo and Search Attributes

What are Memo and Search Attributes?

Memo: User-defined metadata attached to workflow executions for organizational purposes. Memo data is not indexed and is primarily used for workflow identification and organization.

Search Attributes: Indexed metadata that enables efficient querying and filtering of workflow executions. Search attributes are optimized for fast retrieval and are essential for building dashboards and monitoring systems.

Using Memo

```
from temporalio import workflow
from datetime import datetime

@workflow.defn
class MemoWorkflow:
    @workflow.run
    async def run(self, input_data: dict) -> str:
        # Memo is set when starting the workflow
        # This is typically done by the client
        return "Workflow completed"

# Client-side memo usage
async def start_workflow_with_memo():
    client = await Client.connect("localhost:7233")

    # Start workflow with memo
    handle = await client.start_workflow(
        MemoWorkflow.run,
        {"test": "data"},
        id="workflow-with-memo",
        memo={
```

```
            "customer_id": "cust_123",
            "order_type": "premium",
            "created_by": "user@example.com",
            "priority": "high"
        }
    )

    result = await handle.result()
    return result
```

Using Search Attributes

```
from temporalio import workflow
from temporalio.common import SearchAttribute

@workflow.defn
class SearchableWorkflow:
    @workflow.run
    async def run(self, input_data: dict) -> str:
        # Search attributes are set when starting the workflow
        # This is typically done by the client
        return "Workflow completed"

# Client-side search attributes usage
async def start_workflow_with_search_attributes():
    client = await Client.connect("localhost:7233")

    # Start workflow with search attributes
    handle = await client.start_workflow(
        SearchableWorkflow.run,
        {"test": "data"},
        id="searchable-workflow",
        search_attributes={
            SearchAttribute.CUSTOM_STRING_FIELD_1: "customer_123",
            SearchAttribute.CUSTOM_INT_FIELD_1: 1000,
            SearchAttribute.CUSTOM_DOUBLE_FIELD_1: 99.99,
            SearchAttribute.CUSTOM_BOOL_FIELD_1: True,
            SearchAttribute.CUSTOM_DATETIME_FIELD_1: datetime.utcnow()
        }
    )

    result = await handle.result()
    return result
```

Querying Workflows with Search Attributes

```
async def query_workflows_by_search_attributes():
    client = await Client.connect("localhost:7233")
```

```
    # Query workflows by search attributes
    workflows = await client.list_workflows(
        query="SearchAttributes.CustomStringField1 = 'customer_123' AND
SearchAttributes.CustomIntField1 > 500"
    )

    for workflow in workflows:
        print(f"Workflow ID: {workflow.id}")
        print(f"Status: {workflow.status}")
        print(f"Search Attributes: {workflow.search_attributes}")
```

Advanced Search Attribute Patterns

```
# Search attribute constants for consistency
class WorkflowSearchAttributes:
    CUSTOMER_ID = SearchAttribute.CUSTOM_STRING_FIELD_1
    ORDER_AMOUNT = SearchAttribute.CUSTOM_DOUBLE_FIELD_1
    PRIORITY = SearchAttribute.CUSTOM_INT_FIELD_1
    IS_PREMIUM = SearchAttribute.CUSTOM_BOOL_FIELD_1
    CREATED_DATE = SearchAttribute.CUSTOM_DATETIME_FIELD_1

# Workflow with structured search attributes
@workflow.defn
class EcommerceWorkflow:
    @workflow.run
    async def run(self, order_data: dict) -> dict:
        # Workflow logic here
        return {"order_id": order_data["order_id"], "status": "completed"}

# Client helper for consistent search attributes
class WorkflowClient:
    def __init__(self, client):
        self.client = client

    async def start_ecommerce_workflow(self, order_data: dict):
        search_attrs = {
            WorkflowSearchAttributes.CUSTOMER_ID:
order_data["customer_id"],
            WorkflowSearchAttributes.ORDER_AMOUNT: order_data["amount"],
            WorkflowSearchAttributes.PRIORITY: order_data.get("priority",
1),
            WorkflowSearchAttributes.IS_PREMIUM:
order_data.get("is_premium", False),
            WorkflowSearchAttributes.CREATED_DATE: datetime.utcnow()
        }

        return await self.client.start_workflow(
            EcommerceWorkflow.run,
            order_data,
            id=f"order-{order_data['order_id']}",
            search_attributes=search_attrs
```

```
        )

    async def find_customer_orders(self, customer_id: str):
        """Find all workflows for a specific customer."""
        return await self.client.list_workflows(
            query=f"SearchAttributes.CustomStringField1 = '{customer_id}'"
        )

    async def find_high_value_orders(self, min_amount: float):
        """Find all high-value orders."""
        return await self.client.list_workflows(
            query=f"SearchAttributes.CustomDoubleField1 >= {min_amount}"
        )
```

Implementing Custom Data Converters

Why Custom Data Converters?

Temporal's default data converter uses JSON serialization, which has limitations:

1. **Type Information Loss:** JSON doesn't preserve Python types
2. **Performance:** JSON serialization can be slow for large objects
3. **Security:** JSON doesn't support encryption
4. **Custom Types:** Complex objects may not serialize properly

Basic Custom Data Converter

```
from temporalio.converter import DataConverter, DefaultDataConverter
import pickle
import base64

class PickleDataConverter(DataConverter):
    """Custom data converter using pickle for serialization."""

    def __init__(self):
        self.payload_converter_class = PicklePayloadConverter

class PicklePayloadConverter:
    """Payload converter that uses pickle."""

    def to_payload(self, value) -> bytes:
        """Convert Python object to bytes."""
        if value is None:
            return b""
        return pickle.dumps(value)

    def from_payload(self, payload: bytes):
        """Convert bytes back to Python object."""
        if not payload:
            return None
```

```
        return pickle.loads(payload)

# Usage
pickle_converter = PickleDataConverter()

# Start workflow with custom converter
handle = await client.start_workflow(
    MyWorkflow.run,
    complex_object,
    id="workflow-with-pickle",
    data_converter=pickle_converter
)
```

Encrypted Data Converter

```
from cryptography.fernet import Fernet
import json

class EncryptedDataConverter(DataConverter):
    """Data converter with encryption."""

    def __init__(self, encryption_key: bytes):
        self.fernet = Fernet(encryption_key)
        self.payload_converter_class = EncryptedPayloadConverter

class EncryptedPayloadConverter:
    """Payload converter with encryption."""

    def __init__(self, fernet: Fernet):
        self.fernet = fernet

    def to_payload(self, value) -> bytes:
        """Encrypt and serialize value."""
        if value is None:
            return b""

        # Serialize to JSON
        json_data = json.dumps(value, default=str)

        # Encrypt
        encrypted_data = self.fernet.encrypt(json_data.encode())

        return encrypted_data

    def from_payload(self, payload: bytes):
        """Decrypt and deserialize value."""
        if not payload:
            return None

        # Decrypt
        decrypted_data = self.fernet.decrypt(payload)
```

```
        # Deserialize from JSON
        return json.loads(decrypted_data.decode())

# Usage
encryption_key = Fernet.generate_key()
encrypted_converter = EncryptedDataConverter(encryption_key)

# Start workflow with encrypted data
handle = await client.start_workflow(
    SecureWorkflow.run,
    sensitive_data,
    id="secure-workflow",
    data_converter=encrypted_converter
)
```

Compressed Data Converter

```
import gzip
import json

class CompressedDataConverter(DataConverter):
    """Data converter with compression."""

    def __init__(self):
        self.payload_converter_class = CompressedPayloadConverter

class CompressedPayloadConverter:
    """Payload converter with compression."""

    def to_payload(self, value) -> bytes:
        """Compress and serialize value."""
        if value is None:
            return b""

        # Serialize to JSON
        json_data = json.dumps(value, default=str)

        # Compress
        compressed_data = gzip.compress(json_data.encode())

        return compressed_data

    def from_payload(self, payload: bytes):
        """Decompress and deserialize value."""
        if not payload:
            return None

        # Decompress
        decompressed_data = gzip.decompress(payload)
```

```
        # Deserialize from JSON
        return json.loads(decompressed_data.decode())

# Usage
compressed_converter = CompressedDataConverter()

# Start workflow with compressed data
handle = await client.start_workflow(
    LargeDataWorkflow.run,
    large_dataset,
    id="compressed-workflow",
    data_converter=compressed_converter
)
```

Advanced Workflow Patterns

Pattern 1: Saga Pattern with Compensation

```
@workflow.defn
class SagaWorkflow:
    def __init__(self):
        self.completed_steps = []
        self.compensation_required = False

    @workflow.run
    async def run(self, saga_data: dict) -> dict:
        try:
            # Step 1: Reserve inventory
            inventory_result = await workflow.execute_activity(
                reserve_inventory_activity,
                saga_data["items"],
                start_to_close_timeout=timedelta(seconds=30)
            )
            self.completed_steps.append("inventory")

            # Step 2: Process payment
            payment_result = await workflow.execute_activity(
                process_payment_activity,
                saga_data["payment_info"],
                start_to_close_timeout=timedelta(seconds=30)
            )
            self.completed_steps.append("payment")

            # Step 3: Ship order
            shipping_result = await workflow.execute_activity(
                ship_order_activity,
                saga_data["shipping_info"],
                start_to_close_timeout=timedelta(seconds=60)
            )
            self.completed_steps.append("shipping")
```

```
            return {
                "status": "completed",
                "inventory_id": inventory_result["inventory_id"],
                "payment_id": payment_result["payment_id"],
                "tracking_number": shipping_result["tracking_number"]
            }

        except Exception as e:
            workflow.logger.error(f"Saga failed: {e}")
            self.compensation_required = True
            await self._compensate()
            raise e

    async def _compensate(self):
        """Compensate for completed steps in reverse order."""
        for step in reversed(self.completed_steps):
            try:
                if step == "shipping":
                    await workflow.execute_activity(
                        cancel_shipping_activity,
                        start_to_close_timeout=timedelta(seconds=30)
                    )
                elif step == "payment":
                    await workflow.execute_activity(
                        refund_payment_activity,
                        start_to_close_timeout=timedelta(seconds=30)
                    )
                elif step == "inventory":
                    await workflow.execute_activity(
                        release_inventory_activity,
                        start_to_close_timeout=timedelta(seconds=30)
                    )
            except Exception as e:
                workflow.logger.error(f"Compensation failed for {step}:
{e}")
```

Pattern 2: Event Sourcing with Temporal

```
@workflow.defn
class EventSourcedWorkflow:
    def __init__(self):
        self.events = []
        self.state = {}

    @workflow.run
    async def run(self, initial_state: dict) -> dict:
        self.state = initial_state.copy()

        # Apply initial state
        self._apply_event("StateInitialized", initial_state)
```

```
        # Main event processing loop
        while True:
            # Wait for external events
            await workflow.wait_condition(lambda: len(self.events) > 0)

            # Process events
            for event in self.events:
                await self._process_event(event)

            # Clear processed events
            self.events.clear()

    def _apply_event(self, event_type: str, event_data: dict):
        """Apply an event to the state."""
        event = {
            "id": len(self.events),
            "type": event_type,
            "data": event_data,
            "timestamp": workflow.now()
        }

        self.events.append(event)

        # Update state based on event type
        if event_type == "StateInitialized":
            self.state.update(event_data)
        elif event_type == "OrderPlaced":
            self.state["orders"] = self.state.get("orders", []) +
[event_data]
        elif event_type == "PaymentProcessed":
            self.state["payments"] = self.state.get("payments", []) +
[event_data]

    async def _process_event(self, event: dict):
        """Process an event and trigger side effects."""
        if event["type"] == "OrderPlaced":
            # Trigger payment processing
            await workflow.execute_activity(
                process_payment_activity,
                event["data"],
                start_to_close_timeout=timedelta(seconds=30)
            )
        elif event["type"] == "PaymentProcessed":
            # Trigger shipping
            await workflow.execute_activity(
                ship_order_activity,
                event["data"],
                start_to_close_timeout=timedelta(seconds=60)
            )

    @workflow.signal
    async def place_order(self, order_data: dict):
        """Signal to place an order."""
```

```
        self._apply_event("OrderPlaced", order_data)

    @workflow.signal
    async def process_payment(self, payment_data: dict):
        """Signal to process payment."""
        self._apply_event("PaymentProcessed", payment_data)

    @workflow.query
    def get_state(self) -> dict:
        """Query current state."""
        return self.state.copy()
```

Pattern 3: Circuit Breaker with Temporal

```
@workflow.defn
class CircuitBreakerWorkflow:
    def __init__(self):
        self.failure_count = 0
        self.last_failure_time = None
        self.circuit_state = "CLOSED"  # CLOSED, OPEN, HALF_OPEN
        self.threshold = 5
        self.timeout = timedelta(minutes=5)

    @workflow.run
    async def run(self, operation_data: dict) -> dict:
        while True:
            if self.circuit_state == "CLOSED":
                result = await self._try_operation(operation_data)
                if result:
                    return result
            elif self.circuit_state == "OPEN":
                # Wait for timeout
                await workflow.sleep(self.timeout.total_seconds())
                self.circuit_state = "HALF_OPEN"
            elif self.circuit_state == "HALF_OPEN":
                # Try once
                result = await self._try_operation(operation_data)
                if result:
                    self.circuit_state = "CLOSED"
                    self.failure_count = 0
                    return result
                else:
                    self.circuit_state = "OPEN"

    async def _try_operation(self, operation_data: dict) -> dict:
        """Try to execute the protected operation."""
        try:
            result = await workflow.execute_activity(
                protected_operation_activity,
                operation_data,
                start_to_close_timeout=timedelta(seconds=30)
```

```
            )

            # Success - reset failure count
            self.failure_count = 0
            return result

        except Exception as e:
            # Failure - increment failure count
            self.failure_count += 1
            self.last_failure_time = workflow.now()

            # Check if circuit should open
            if self.failure_count >= self.threshold:
                self.circuit_state = "OPEN"
                workflow.logger.warning("Circuit breaker opened")

            return None

    @workflow.query
    def get_circuit_status(self) -> dict:
        """Get current circuit breaker status."""
        return {
            "state": self.circuit_state,
            "failure_count": self.failure_count,
            "last_failure_time": self.last_failure_time,
            "threshold": self.threshold
        }
```

Real-World Example: Dynamic Activity Selection

E-commerce Order Processing with Dynamic Activities

```
@workflow.defn
class DynamicOrderWorkflow:
    @workflow.run
    async def run(self, order_data: dict) -> dict:
        # Determine processing strategy based on order type
        strategy = self._determine_strategy(order_data)

        # Select activities based on strategy
        activities = self._select_activities(strategy)

        # Execute selected activities
        results = {}
        for activity_name, activity_func in activities.items():
            try:
                result = await workflow.execute_activity(
                    activity_func,
                    order_data,
                    start_to_close_timeout=timedelta(seconds=60)
```

```
                )
                results[activity_name] = result
            except Exception as e:
                workflow.logger.error(f"Activity {activity_name} failed:
{e}")
                results[activity_name] = {"error": str(e)}

        return {
            "order_id": order_data["order_id"],
            "strategy": strategy,
            "results": results
        }

    def _determine_strategy(self, order_data: dict) -> str:
        """Determine processing strategy based on order characteristics."""
        if order_data.get("is_premium"):
            return "premium"
        elif order_data.get("amount", 0) > 1000:
            return "high_value"
        elif order_data.get("is_international"):
            return "international"
        else:
            return "standard"

    def _select_activities(self, strategy: str) -> dict:
        """Select activities based on strategy."""
        activity_registry = DynamicActivityRegistry()

        if strategy == "premium":
            return {
                "fraud_detection":
activity_registry.get_activity("premium_fraud_detection"),
                "payment_processing":
activity_registry.get_activity("premium_payment_processing"),
                "shipping":
activity_registry.get_activity("express_shipping"),
                "notification":
activity_registry.get_activity("premium_notification")
            }
        elif strategy == "high_value":
            return {
                "fraud_detection":
activity_registry.get_activity("enhanced_fraud_detection"),
                "payment_processing":
activity_registry.get_activity("secure_payment_processing"),
                "shipping":
activity_registry.get_activity("insured_shipping"),
                "notification":
activity_registry.get_activity("high_value_notification")
            }
        elif strategy == "international":
            return {
                "fraud_detection":
activity_registry.get_activity("international_fraud_detection"),
```

```
                "payment_processing":
activity_registry.get_activity("international_payment_processing"),
                "customs":
activity_registry.get_activity("customs_processing"),
                "shipping":
activity_registry.get_activity("international_shipping"),
                "notification":
activity_registry.get_activity("international_notification")
            }
        else:  # standard
            return {
                "fraud_detection":
activity_registry.get_activity("basic_fraud_detection"),
                "payment_processing":
activity_registry.get_activity("standard_payment_processing"),
                "shipping":
activity_registry.get_activity("standard_shipping"),
                "notification":
activity_registry.get_activity("standard_notification")
            }

# Activity implementations
@activity.defn
async def premium_fraud_detection(order_data: dict) -> dict:
    # Enhanced fraud detection for premium orders
    return {"fraud_score": 0.1, "approved": True}

@activity.defn
async def enhanced_fraud_detection(order_data: dict) -> dict:
    # Enhanced fraud detection for high-value orders
    return {"fraud_score": 0.05, "approved": True}

@activity.defn
async def international_fraud_detection(order_data: dict) -> dict:
    # International fraud detection
    return {"fraud_score": 0.2, "approved": True}

@activity.defn
async def basic_fraud_detection(order_data: dict) -> dict:
    # Basic fraud detection
    return {"fraud_score": 0.3, "approved": True}

# Similar implementations for other activities...
```

Performance Optimization Techniques

1. Activity Batching

```
@workflow.defn
class BatchedWorkflow:
```

```
    @workflow.run
    async def run(self, items: list) -> dict:
        # Process items in batches
        batch_size = 100
        results = []

        for i in range(0, len(items), batch_size):
            batch = items[i:i + batch_size]

            # Single activity call for entire batch
            batch_result = await workflow.execute_activity(
                process_batch_activity,
                batch,
                start_to_close_timeout=timedelta(minutes=30)
            )
            results.extend(batch_result)

        return {"processed_items": len(results)}
```

2. Parallel Activity Execution

```
@workflow.defn
class ParallelWorkflow:
    @workflow.run
    async def run(self, data: dict) -> dict:
        # Execute activities in parallel
        tasks = [
            workflow.execute_activity(activity_1, data),
            workflow.execute_activity(activity_2, data),
            workflow.execute_activity(activity_3, data)
        ]

        results = await asyncio.gather(*tasks)

        return {
            "result_1": results[0],
            "result_2": results[1],
            "result_3": results[2]
        }
```

3. Caching with Memo

```
@workflow.defn
class CachedWorkflow:
    @workflow.run
    async def run(self, input_data: dict) -> dict:
        # Check if we have cached result
        cache_key = self._generate_cache_key(input_data)
```

```
        # Try to get from cache first
        cached_result = await workflow.execute_activity(
            get_from_cache_activity,
            cache_key,
            start_to_close_timeout=timedelta(seconds=5)
        )

        if cached_result:
            return cached_result

        # Compute result
        result = await workflow.execute_activity(
            expensive_computation_activity,
            input_data,
            start_to_close_timeout=timedelta(minutes=10)
        )

        # Cache result
        await workflow.execute_activity(
            store_in_cache_activity,
            cache_key,
            result,
            start_to_close_timeout=timedelta(seconds=5)
        )

        return result

    def _generate_cache_key(self, data: dict) -> str:
        """Generate cache key from input data."""
        return f"workflow_cache_{hash(str(data))}"
```

Security and Compliance Patterns

1. Data Masking

```
class DataMaskingConverter(DataConverter):
    """Data converter that masks sensitive information."""

    def __init__(self):
        self.payload_converter_class = DataMaskingPayloadConverter

class DataMaskingPayloadConverter:
    """Payload converter with data masking."""

    def to_payload(self, value) -> bytes:
        """Mask sensitive data before serialization."""
        if isinstance(value, dict):
            masked_value = self._mask_dict(value)
        else:
            masked_value = value
```

```
        return json.dumps(masked_value, default=str).encode()

    def from_payload(self, payload: bytes):
        """Deserialize payload."""
        if not payload:
            return None
        return json.loads(payload.decode())

    def _mask_dict(self, data: dict) -> dict:
        """Mask sensitive fields in dictionary."""
        masked = data.copy()

        sensitive_fields = ["password", "credit_card", "ssn", "api_key"]

        for field in sensitive_fields:
            if field in masked:
                masked[field] = "***MASKED***"

        return masked
```

2. Audit Logging

```
@workflow.defn
class AuditedWorkflow:
    @workflow.run
    async def run(self, input_data: dict) -> dict:
        # Log workflow start
        await self._audit_log("WORKFLOW_STARTED", input_data)

        try:
            # Execute business logic
            result = await workflow.execute_activity(
                business_logic_activity,
                input_data,
                start_to_close_timeout=timedelta(seconds=30)
            )

            # Log successful completion
            await self._audit_log("WORKFLOW_COMPLETED", result)

            return result

        except Exception as e:
            # Log failure
            await self._audit_log("WORKFLOW_FAILED", {"error": str(e)})
            raise e

    async def _audit_log(self, event_type: str, data: dict):
        """Log audit event."""
        audit_event = {
```

```
            "workflow_id": workflow.info().workflow_id,
            "event_type": event_type,
            "timestamp": workflow.now(),
            "data": data
        }

        await workflow.execute_activity(
            log_audit_event_activity,
            audit_event,
            start_to_close_timeout=timedelta(seconds=10)
        )
```

Best Practices and Anti-Patterns

Best Practices

1. **Use Descriptive Names:** Use clear, descriptive names for activities and workflows
2. **Implement Proper Error Handling:** Always handle errors gracefully
3. **Use Appropriate Timeouts:** Set realistic timeouts for activities
4. **Monitor Performance:** Track performance metrics for workflows
5. **Document Complex Logic:** Document complex workflow logic thoroughly

Anti-Patterns

1. **Long-Running Activities:** Don't make activities run for hours
2. **Complex Workflow Logic:** Keep workflow logic simple and focused
3. **Ignoring Errors:** Don't ignore or suppress errors
4. **Hard-Coded Values:** Avoid hard-coding configuration values

---

Testing Advanced Patterns

Unit Testing Dynamic Registration

```
def test_dynamic_activity_registration():
    registry = DynamicActivityRegistry()

    # Register test activity
    @activity.defn
    def test_activity(data):
        return {"processed": data}

    registry.register_activity("test_activity", test_activity)

    # Test retrieval
    retrieved_activity = registry.get_activity("test_activity")
    assert retrieved_activity == test_activity
```

```
    # Test non-existent activity
    assert registry.get_activity("non_existent") is None
```

Integration Testing with Custom Converters

```
async def test_encrypted_data_converter():
    # Generate encryption key
    key = Fernet.generate_key()
    converter = EncryptedDataConverter(key)

    # Test data
    test_data = {"sensitive": "information", "amount": 100}

    # Test serialization/deserialization
    payload =
converter.payload_converter_class(converter.fernet).to_payload(test_data)
    deserialized =
converter.payload_converter_class(converter.fernet).from_payload(payload)

    assert deserialized == test_data
```

Troubleshooting Advanced Issues

Common Issues and Solutions

1. **Dynamic Registration Failures:**
   - Ensure activities are properly decorated
   - Check module import paths
   - Verify activity function signatures
2. **Search Attribute Issues:**
   - Ensure search attributes are properly indexed
   - Check query syntax
   - Verify attribute types match
3. **Custom Converter Problems:**
   - Test serialization/deserialization thoroughly
   - Handle None values properly
   - Ensure backward compatibility
4. **Performance Issues:**
   - Monitor activity execution times
   - Use batching for large datasets
   - Implement caching where appropriate

Summary

Today, we explored advanced workflow patterns and best practices in Temporal:

- Dynamic activity and workflow registration for flexible systems
- Using memo and search attributes for enhanced visibility and querying
- Implementing custom data converters for complex payload serialization
- Advanced workflow patterns including Saga, Event Sourcing, and Circuit Breaker
- Performance optimization techniques and security patterns
- Best practices for building production-grade Temporal applications

Key Takeaways:

- Use dynamic registration for flexible, plugin-based systems
- Leverage search attributes for efficient workflow querying
- Implement custom converters for specific serialization needs
- Apply advanced patterns for complex business scenarios
- Focus on performance, security, and maintainability

Tomorrow, we'll explore testing strategies for Temporal workflows and activities, ensuring your applications are robust and reliable.