Temporal Training Session 9: Workflow Versioning and Long-Running Flows

Table of Contents

Introduction and Recap

Welcome to Day 9 of our Temporal training. Yesterday, we explored child workflows and parallel processing patterns—powerful tools for building scalable, distributed systems. Today, we tackle one of the most critical aspects of production Temporal applications: **workflow versioning and long-running flows**.

Learning Objectives:

- Understand why workflow versioning is essential for determinism and upgrades
- Learn to use `Workflow.get_version()` safely for logic migration
- Implement strategies for migrating logic in live workflows
- Design and manage long-running workflows spanning days or months
- Understand history size and performance considerations

Why Workflow Versioning is Essential

The Challenge of Determinism

Temporal workflows must be deterministic—given the same input and event history, they must produce the same output. This requirement creates a unique challenge when you need to update workflow logic while workflows are still running.

The Problem with Direct Updates

If you simply update workflow code and deploy it:

1. **New workflows** will use the new logic
2. **Existing workflows** will continue using the old logic when they replay
3. **This can lead to inconsistencies** and unexpected behavior

Example: The Versioning Problem

```python
# Version 1 of a workflow
@workflow.defn
class PaymentWorkflow:
    @workflow.run
    async def run(self, amount: float) -> str:
        # Old logic: simple payment processing
        result = await workflow.execute_activity(
            process_payment_activity,
            amount,
            start_to_close_timeout=timedelta(seconds=30)
        )
        return f"Payment processed: {result}"

# Later, you want to add fraud detection
# Version 2 (naive approach - DON'T DO THIS)
@workflow.defn
class PaymentWorkflow:
    @workflow.run
    async def run(self, amount: float) -> str:
        # New logic: add fraud detection
        fraud_check = await workflow.execute_activity(
            fraud_detection_activity,
            amount,
            start_to_close_timeout=timedelta(seconds=60)
        )

        if fraud_check['suspicious']:
            return "Payment flagged for review"

        # Continue with payment processing
        result = await workflow.execute_activity(
            process_payment_activity,
            amount,
            start_to_close_timeout=timedelta(seconds=30)
        )
        return f"Payment processed: {result}"
```

The Problem: Existing workflows that were started with Version 1 will fail when they replay because the new logic expects a fraud check step that wasn't in the original execution.

Understanding Determinism and Upgrades

How Temporal Ensures Determinism

Temporal uses event sourcing to maintain workflow state. Each workflow execution has a complete history of events:

1. **WorkflowExecutionStarted** - Initial event with input
2. **ActivityTaskScheduled** - When an activity is scheduled
3. **ActivityTaskCompleted** - When an activity completes
4. **TimerStarted** - When a timer is started
5. **TimerFired** - When a timer fires
6. **WorkflowExecutionCompleted** - Final event

The Replay Process

When a workflow needs to resume (after a worker restart, for example):

1. Temporal loads the complete event history
2. It replays the workflow function from the beginning
3. The SDK skips commands that are already in the history
4. The workflow continues from where it left off

Why This Breaks with Direct Updates

If you change the workflow logic and deploy:

- **New workflows** start with the new logic
- **Existing workflows** replay with the new logic but expect the old event history
- **Mismatch occurs** when the new logic tries to execute steps that weren't in the original history

---

Using Workflow.get_version() Safely

The Versioning Solution

Temporal provides `workflow.get_version()` to handle logic migrations safely. This function allows you to:

1. **Detect the current version** of a workflow
2. **Execute different logic** based on the version
3. **Maintain determinism** across deployments

Basic Versioning Pattern

```
@workflow.defn
class VersionedWorkflow:
    @workflow.run
    async def run(self, input_data: dict) -> str:
        # Get the current version of this workflow
        version = workflow.get_version("payment-processing", 1, 2)

        if version == 1:
            # Version 1 logic (original)
```

```
            result = await workflow.execute_activity(
                process_payment_activity,
                input_data,
                start_to_close_timeout=timedelta(seconds=30)
            )
            return f"Payment processed (v1): {result}"

        elif version == 2:
            # Version 2 logic (with fraud detection)
            fraud_check = await workflow.execute_activity(
                fraud_detection_activity,
                input_data,
                start_to_close_timeout=timedelta(seconds=60)
            )

            if fraud_check['suspicious']:
                return "Payment flagged for review (v2)"

            result = await workflow.execute_activity(
                process_payment_activity,
                input_data,
                start_to_close_timeout=timedelta(seconds=30)
            )
            return f"Payment processed (v2): {result}"
```

How get_version() Works

```
workflow.get_version(change_id, min_supported_version,
max_supported_version)
```

- **change_id:** A unique identifier for this change
- **min_supported_version:** The minimum version that supports this change
- **max_supported_version:** The maximum version that supports this change

Return Value:

- Returns the version number that should be used for this workflow execution
- The version is deterministic based on the workflow's execution history

Multiple Version Points

You can have multiple versioning points in a single workflow:

```
@workflow.defn
class ComplexVersionedWorkflow:
    @workflow.run
    async def run(self, input_data: dict) -> str:
        # Version point 1: Payment processing
        payment_version = workflow.get_version("payment-processing", 1, 2)
```

```
        if payment_version == 1:
            payment_result = await self._process_payment_v1(input_data)
        else:
            payment_result = await self._process_payment_v2(input_data)

        # Version point 2: Notification system
        notification_version = workflow.get_version("notification-system",
1, 3)

        if notification_version == 1:
            await self._send_notification_v1(payment_result)
        elif notification_version == 2:
            await self._send_notification_v2(payment_result)
        else:
            await self._send_notification_v3(payment_result)

        return payment_result

    async def _process_payment_v1(self, input_data: dict) -> str:
        # Version 1 payment logic
        pass

    async def _process_payment_v2(self, input_data: dict) -> str:
        # Version 2 payment logic with fraud detection
        pass
```

Strategies for Migrating Logic in Live Workflows

Strategy 1: Gradual Migration

Deploy new versions gradually, allowing old workflows to complete naturally:

```
@workflow.defn
class GradualMigrationWorkflow:
    @workflow.run
    async def run(self, input_data: dict) -> str:
        version = workflow.get_version("gradual-migration", 1, 2)

        if version == 1:
            # Old logic - let existing workflows complete
            return await self._old_logic(input_data)
        else:
            # New logic - for new workflows
            return await self._new_logic(input_data)

    async def _old_logic(self, input_data: dict) -> str:
        # Original implementation
        result = await workflow.execute_activity(
            old_activity,
```

```
            input_data,
            start_to_close_timeout=timedelta(seconds=30)
        )
        return f"Old result: {result}"

    async def _new_logic(self, input_data: dict) -> str:
        # New implementation with improvements
        result = await workflow.execute_activity(
            new_activity,
            input_data,
            start_to_close_timeout=timedelta(seconds=45)
        )
        return f"New result: {result}"
```

Strategy 2: Feature Flags

Use versioning to implement feature flags:

```
@workflow.defn
class FeatureFlagWorkflow:
    @workflow.run
    async def run(self, input_data: dict) -> str:
        # Check if new feature is enabled
        new_feature_version = workflow.get_version("new-feature", 1, 2)

        # Common logic
        base_result = await workflow.execute_activity(
            base_activity,
            input_data,
            start_to_close_timeout=timedelta(seconds=30)
        )

        if new_feature_version == 2:
            # Apply new feature
            enhanced_result = await workflow.execute_activity(
                enhancement_activity,
                base_result,
                start_to_close_timeout=timedelta(seconds=20)
            )
            return enhanced_result
        else:
            # Return base result
            return base_result
```

Strategy 3: Backward Compatibility

Ensure new versions can handle old data formats:

```python
@workflow.defn
class BackwardCompatibleWorkflow:
    @workflow.run
    async def run(self, input_data: dict) -> str:
        version = workflow.get_version("data-format", 1, 2)

        if version == 1:
            # Handle old data format
            processed_data = self._process_old_format(input_data)
        else:
            # Handle new data format
            processed_data = self._process_new_format(input_data)

        # Common processing logic
        result = await workflow.execute_activity(
            process_activity,
            processed_data,
            start_to_close_timeout=timedelta(seconds=30)
        )

        return result

    def _process_old_format(self, data: dict) -> dict:
        # Convert old format to common format
        return {
            "amount": data.get("payment_amount", 0),
            "currency": data.get("currency", "USD"),
            "customer_id": data.get("user_id")
        }

    def _process_new_format(self, data: dict) -> dict:
        # New format is already in common format
        return data
```

Designing Long-Running Workflows

Characteristics of Long-Running Workflows

Long-running workflows (spanning days, weeks, or months) have unique characteristics:

1. **State Persistence:** State must be maintained across long periods
2. **External Dependencies:** Often depend on external events or human input
3. **Complex Business Logic:** May involve multiple decision points
4. **Resource Management:** Need to manage resources over extended periods

Example: Subscription Management Workflow

```python
@workflow.defn
class SubscriptionWorkflow:
```

```
    def __init__(self):
        self.subscription_status = "active"
        self.last_billing_date = None
        self.next_billing_date = None
        self.payment_failures = 0

    @workflow.run
    async def run(self, customer_id: str, plan_details: dict) -> dict:
        workflow.logger.info(f"Starting subscription workflow for customer
{customer_id}")

        # Initialize subscription
        await self._initialize_subscription(customer_id, plan_details)

        # Main subscription loop
        while self.subscription_status == "active":
            # Wait for next billing cycle
            await self._wait_for_billing_cycle()

            # Process billing
            billing_success = await self._process_billing(customer_id,
plan_details)

            if not billing_success:
                await self._handle_billing_failure(customer_id)

            # Check for cancellation signals
            if self._should_cancel():
                break

        return await self._finalize_subscription(customer_id)

    async def _initialize_subscription(self, customer_id: str,
plan_details: dict):
        # Set up initial billing date
        self.next_billing_date = workflow.now() + timedelta(days=30)

        # Send welcome notification
        await workflow.execute_activity(
            send_welcome_notification_activity,
            customer_id,
            start_to_close_timeout=timedelta(seconds=30)
        )

    async def _wait_for_billing_cycle(self):
        # Calculate time until next billing
        time_until_billing = (self.next_billing_date -
workflow.now()).total_seconds()

        if time_until_billing > 0:
            await workflow.sleep(time_until_billing)

        # Update billing date
        self.last_billing_date = self.next_billing_date
```

```
        self.next_billing_date = self.last_billing_date +
timedelta(days=30)

    async def _process_billing(self, customer_id: str, plan_details: dict)
-> bool:
        try:
            # Attempt to charge the customer
            billing_result = await workflow.execute_activity(
                charge_customer_activity,
                customer_id,
                plan_details['amount'],
                start_to_close_timeout=timedelta(seconds=60)
            )

            if billing_result['success']:
                self.payment_failures = 0

                # Send successful billing notification
                await workflow.execute_activity(
                    send_billing_success_activity,
                    customer_id,
                    billing_result,
                    start_to_close_timeout=timedelta(seconds=30)
                )

                return True
            else:
                self.payment_failures += 1
                return False

        except Exception as e:
            workflow.logger.error(f"Billing failed for customer
{customer_id}: {e}")
            self.payment_failures += 1
            return False

    async def _handle_billing_failure(self, customer_id: str):
        if self.payment_failures >= 3:
            # Suspend subscription after 3 failures
            self.subscription_status = "suspended"

            await workflow.execute_activity(
                suspend_subscription_activity,
                customer_id,
                start_to_close_timeout=timedelta(seconds=30)
            )
        else:
            # Send failure notification
            await workflow.execute_activity(
                send_billing_failure_activity,
                customer_id,
                self.payment_failures,
                start_to_close_timeout=timedelta(seconds=30)
            )
```

```
    def _should_cancel(self) -> bool:
        # Check for cancellation signal
        return self.subscription_status == "cancelled"

    async def _finalize_subscription(self, customer_id: str) -> dict:
        # Clean up subscription
        await workflow.execute_activity(
            cleanup_subscription_activity,
            customer_id,
            start_to_close_timeout=timedelta(seconds=30)
        )

        return {
            "customer_id": customer_id,
            "final_status": self.subscription_status,
            "total_billing_cycles": self.payment_failures
        }

    @workflow.signal
    async def cancel_subscription(self):
        self.subscription_status = "cancelled"
        workflow.logger.info("Subscription cancellation requested")
```

History Size and Performance Considerations

Understanding History Growth

Workflow history grows with each event. For long-running workflows, this can become significant:

Events that add to history:

- Activity scheduling and completion
- Timer creation and firing
- Signal reception
- Child workflow operations
- Version changes

Monitoring History Size

```
@workflow.defn
class HistoryMonitoringWorkflow:
    @workflow.run
    async def run(self, input_data: dict) -> dict:
        # Check history size periodically
        while True:
            # Get workflow info
            info = workflow.info()

            # Log history size
```

```
            workflow.logger.info(f"History size: {info.history_length}
events")

            # If history is getting large, consider cleanup
            if info.history_length > 1000:
                workflow.logger.warning("History size is large, consider
optimization")

            # Wait before next check
            await workflow.sleep(3600)  # Check every hour
```

Strategies for Managing History Size

1. Use Activities for Heavy Processing

```
# Good: Move heavy processing to activities
@workflow.defn
class OptimizedWorkflow:
    @workflow.run
    async def run(self, data: list) -> dict:
        # Process data in activities to reduce workflow complexity
        result = await workflow.execute_activity(
            heavy_processing_activity,
            data,
            start_to_close_timeout=timedelta(hours=2)
        )
        return result

# Bad: Complex logic in workflow
@workflow.defn
class UnoptimizedWorkflow:
    @workflow.run
    async def run(self, data: list) -> dict:
        # Complex processing in workflow creates many events
        processed_data = []
        for item in data:
            # Each iteration adds events to history
            processed_item = await self._complex_processing(item)
            processed_data.append(processed_item)
        return processed_data
```

2. Batch Operations

```
@workflow.defn
class BatchedWorkflow:
    @workflow.run
    async def run(self, items: list) -> dict:
        # Process items in batches to reduce history events
```

```
        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)}
```

3. Use Child Workflows for Complex Logic

```
@workflow.defn
class ParentWorkflow:
    @workflow.run
    async def run(self, complex_task: dict) -> dict:
        # Delegate complex processing to child workflow
        child_handle = await workflow.start_child_workflow(
            ComplexProcessingWorkflow.run,
            complex_task,
            id=f"complex-{complex_task['id']}"
        )

        result = await child_handle
        return result
```

Real-World Example: Subscription Renewal and Billing

Complete Subscription Management System

```
@workflow.defn
class SubscriptionManagementWorkflow:
    def __init__(self):
        self.subscription_data = {}
        self.billing_history = []
        self.notification_preferences = {}

    @workflow.run
    async def run(self, customer_id: str, subscription_config: dict) ->
dict:
        # Initialize subscription
        await self._setup_subscription(customer_id, subscription_config)
```

```
        # Main subscription lifecycle
        while self.subscription_data['status'] == 'active':
            # Wait for next billing cycle
            await self._wait_for_billing_cycle()

            # Process billing with versioning
            await self._process_billing_with_versioning(customer_id)

            # Check for subscription changes
            await self._handle_subscription_changes(customer_id)

        return await self._finalize_subscription(customer_id)

    async def _setup_subscription(self, customer_id: str, config: dict):
        self.subscription_data = {
            'customer_id': customer_id,
            'plan_type': config['plan_type'],
            'billing_cycle': config['billing_cycle'],
            'amount': config['amount'],
            'status': 'active',
            'start_date': workflow.now(),
            'next_billing_date': workflow.now() +
timedelta(days=config['billing_cycle']),
            'payment_method': config['payment_method']
        }

        # Send welcome notification
        await workflow.execute_activity(
            send_welcome_email_activity,
            customer_id,
            self.subscription_data,
            start_to_close_timeout=timedelta(seconds=30)
        )

    async def _wait_for_billing_cycle(self):
        time_until_billing = (
            self.subscription_data['next_billing_date'] - workflow.now()
        ).total_seconds()

        if time_until_billing > 0:
            await workflow.sleep(time_until_billing)

        # Update billing date
        self.subscription_data['last_billing_date'] =
self.subscription_data['next_billing_date']
        self.subscription_data['next_billing_date'] = (
            self.subscription_data['last_billing_date'] +
            timedelta(days=self.subscription_data['billing_cycle'])
        )

    async def _process_billing_with_versioning(self, customer_id: str):
        # Use versioning for billing logic
        billing_version = workflow.get_version("billing-system", 1, 3)
```

```
        if billing_version == 1:
            await self._process_billing_v1(customer_id)
        elif billing_version == 2:
            await self._process_billing_v2(customer_id)
        else:
            await self._process_billing_v3(customer_id)

    async def _process_billing_v1(self, customer_id: str):
        # Simple billing - just charge the customer
        result = await workflow.execute_activity(
            charge_customer_activity,
            customer_id,
            self.subscription_data['amount'],
            start_to_close_timeout=timedelta(seconds=60)
        )

        self.billing_history.append({
            'date': workflow.now(),
            'amount': self.subscription_data['amount'],
            'success': result['success'],
            'version': 1
        })

    async def _process_billing_v2(self, customer_id: str):
        # Enhanced billing with retry logic
        max_retries = 3
        for attempt in range(max_retries):
            try:
                result = await workflow.execute_activity(
                    charge_customer_activity,
                    customer_id,
                    self.subscription_data['amount'],
                    start_to_close_timeout=timedelta(seconds=60)
                )

                if result['success']:
                    self.billing_history.append({
                        'date': workflow.now(),
                        'amount': self.subscription_data['amount'],
                        'success': True,
                        'attempts': attempt + 1,
                        'version': 2
                    })
                    return
                else:
                    # Wait before retry
                    await workflow.sleep(300)  # 5 minutes

            except Exception as e:
                workflow.logger.error(f"Billing attempt {attempt + 1}
failed: {e}")
                if attempt < max_retries - 1:
                    await workflow.sleep(300)
```

```
        # All attempts failed
        self.billing_history.append({
            'date': workflow.now(),
            'amount': self.subscription_data['amount'],
            'success': False,
            'attempts': max_retries,
            'version': 2
        })

        # Suspend subscription
        self.subscription_data['status'] = 'suspended'

    async def _process_billing_v3(self, customer_id: str):
        # Advanced billing with fraud detection and multiple payment
methods
        # Check for fraud first
        fraud_check = await workflow.execute_activity(
            fraud_detection_activity,
            customer_id,
            self.subscription_data,
            start_to_close_timeout=timedelta(seconds=30)
        )

        if fraud_check['suspicious']:
            self.subscription_data['status'] = 'review'
            return

        # Try primary payment method
        result = await workflow.execute_activity(
            charge_customer_activity,
            customer_id,
            self.subscription_data['amount'],
            self.subscription_data['payment_method'],
            start_to_close_timeout=timedelta(seconds=60)
        )

        if not result['success'] and
self.subscription_data.get('backup_payment_method'):
            # Try backup payment method
            result = await workflow.execute_activity(
                charge_customer_activity,
                customer_id,
                self.subscription_data['amount'],
                self.subscription_data['backup_payment_method'],
                start_to_close_timeout=timedelta(seconds=60)
            )

        self.billing_history.append({
            'date': workflow.now(),
            'amount': self.subscription_data['amount'],
            'success': result['success'],
            'fraud_check': fraud_check,
            'version': 3
```

```
        })

        if not result['success']:
            self.subscription_data['status'] = 'suspended'

    async def _handle_subscription_changes(self, customer_id: str):
        # Check for plan upgrades/downgrades
        # Check for payment method updates
        # Check for cancellation requests
        pass

    async def _finalize_subscription(self, customer_id: str) -> dict:
        # Clean up subscription
        await workflow.execute_activity(
            cleanup_subscription_activity,
            customer_id,
            start_to_close_timeout=timedelta(seconds=30)
        )

        return {
            'customer_id': customer_id,
            'final_status': self.subscription_data['status'],
            'total_billing_cycles': len(self.billing_history),
            'successful_billings': len([b for b in self.billing_history if
b['success']])
        }

    @workflow.signal
    async def update_payment_method(self, new_payment_method: dict):
        self.subscription_data['payment_method'] = new_payment_method
        workflow.logger.info("Payment method updated")

    @workflow.signal
    async def cancel_subscription(self, reason: str):
        self.subscription_data['status'] = 'cancelled'
        self.subscription_data['cancellation_reason'] = reason
        workflow.logger.info(f"Subscription cancelled: {reason}")

    @workflow.query
    def get_subscription_status(self) -> dict:
        return {
            'status': self.subscription_data['status'],
            'next_billing_date':
self.subscription_data['next_billing_date'],
            'billing_history': self.billing_history[-5:]  # Last 5 billing
cycles
        }
```

Advanced Versioning Patterns

Pattern 1: Gradual Feature Rollout

```
@workflow.defn
class GradualRolloutWorkflow:
    @workflow.run
    async def run(self, input_data: dict) -> dict:
        # Check if user is in experimental group
        experimental_version = workflow.get_version("experimental-feature",
1, 2)

        if experimental_version == 2:
            # Use new experimental logic
            return await self._experimental_logic(input_data)
        else:
            # Use stable logic
            return await self._stable_logic(input_data)
```

Pattern 2: A/B Testing

```
@workflow.defn
class ABTestingWorkflow:
    @workflow.run
    async def run(self, user_id: str, input_data: dict) -> dict:
        # Determine A/B test group based on user ID
        test_group = workflow.get_version(f"ab-test-{user_id}", 1, 2)

        if test_group == 1:
            # Group A: Original logic
            return await self._group_a_logic(input_data)
        else:
            # Group B: New logic
            return await self._group_b_logic(input_data)
```

Pattern 3: Regional Rollouts

```
@workflow.defn
class RegionalWorkflow:
    @workflow.run
    async def run(self, region: str, input_data: dict) -> dict:
        # Different versions for different regions
        if region == "us-west":
            version = workflow.get_version("us-west-feature", 1, 2)
        elif region == "eu-west":
            version = workflow.get_version("eu-west-feature", 1, 2)
        else:
            version = workflow.get_version("default-feature", 1, 2)

        if version == 2:
            return await self._new_logic(input_data)
```

```
        else:
            return await self._old_logic(input_data)
```

Best Practices and Anti-Patterns

Best Practices

1. **Plan Versioning Strategy:** Design versioning strategy before implementing changes
2. **Use Descriptive Change IDs:** Use meaningful names for change IDs
3. **Test Versioning Logic:** Thoroughly test versioning logic in development
4. **Monitor Version Distribution:** Track which versions are running in production
5. **Document Changes:** Document all versioning changes and their purposes

Anti-Patterns

1. **Too Many Version Points:** Avoid having too many versioning points in a single workflow
2. **Complex Version Logic:** Keep versioning logic simple and readable
3. **Ignoring Old Versions:** Don't forget to handle old versions properly
4. **Versioning Everything:** Don't version every small change

---

Testing Versioned Workflows

Unit Testing Versioning Logic

```
def test_workflow_versioning():
    # Test that workflow uses correct version
    workflow_instance = VersionedWorkflow()

    # Mock workflow.get_version to return version 1
    with patch('temporalio.workflow.get_version', return_value=1):
        result = workflow_instance.run({"test": "data"})
        assert "v1" in result

    # Mock workflow.get_version to return version 2
    with patch('temporalio.workflow.get_version', return_value=2):
        result = workflow_instance.run({"test": "data"})
        assert "v2" in result
```

Integration Testing

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

    # Start workflow with version 1
    handle = await client.start_workflow(
        VersionedWorkflow.run,
```

```
        {"test": "data"},
        id="test-versioning"
    )

    result = await handle.result()
    assert "v1" in result
```

Troubleshooting Common Issues

Common Issues and Solutions

1. **Version Mismatch Errors:**
    - Ensure version ranges are correct
    - Check that all versions are handled in the code
    - Verify change IDs are unique
2. **History Size Issues:**
    - Monitor history size regularly
    - Use activities for heavy processing
    - Consider breaking large workflows into smaller ones
3. **Performance Issues:**
    - Optimize workflow logic
    - Use appropriate timeouts
    - Monitor resource usage
4. **Versioning Logic Errors:**
    - Test versioning logic thoroughly
    - Use descriptive change IDs
    - Document versioning strategy

Summary

Today, we explored workflow versioning and long-running flows in Temporal:

- Understanding why workflow versioning is essential for determinism and upgrades
- Using `workflow.get_version()` safely for logic migration
- Implementing strategies for migrating logic in live workflows
- Designing and managing long-running workflows
- Understanding history size and performance considerations
- Advanced versioning patterns and best practices

Key Takeaways:

- Always use versioning when updating workflow logic

- Plan versioning strategy before implementing changes
- Monitor history size and performance for long-running workflows
- Test versioning logic thoroughly
- Document all versioning changes

Tomorrow, we'll explore advanced workflow patterns and best practices, building on all the concepts we've learned so far.