Temporal Training Session 8: Child Workflows and Parallel Processing (Fan-out/Fan-in)

Table of Contents

Introduction and Recap

Welcome to Day 8 of our Temporal training. Yesterday, we explored comprehensive error handling and retry strategies—essential for building resilient workflows. Today, we dive into one of Temporal's most powerful features: **child workflows and parallel processing patterns**.

Learning Objectives:

- Understand when and why to use child workflows
- Learn to spawn and manage child workflows from parent workflows
- Implement fan-out/fan-in patterns for parallel processing
- Aggregate results efficiently from multiple child workflows
- Handle errors and implement rollback strategies in distributed processing

---

Understanding Child Workflows

What are Child Workflows?

Child workflows are independent workflow executions that are started by another workflow (the parent). They provide a way to break down complex business logic into smaller, manageable pieces while maintaining the benefits of Temporal's durable execution.

Key Characteristics

- **Independent Execution:** Each child workflow runs independently with its own execution history
- **Fault Isolation:** Failures in child workflows don't automatically fail the parent

- **Scalability:** Child workflows can be distributed across different workers and task queues
- **Modularity:** Complex workflows can be decomposed into smaller, reusable components

Parent-Child Relationship

```python
@workflow.defn
class ParentWorkflow:
    @workflow.run
    async def run(self, input_data: list) -> dict:
        # Parent workflow logic
        results = []
        for item in input_data:
            # Spawn child workflow for each item
            child_handle = await workflow.start_child_workflow(
                ChildWorkflow.run,
                item,
                id=f"child-{item['id']}"
            )
            results.append(child_handle)

        # Wait for all children to complete
        final_results = await asyncio.gather(*results)
        return {"results": final_results}

@workflow.defn
class ChildWorkflow:
    @workflow.run
    async def run(self, item: dict) -> dict:
        # Child workflow logic
        processed_item = await workflow.execute_activity(
            process_item_activity,
            item,
            start_to_close_timeout=timedelta(seconds=30)
        )
        return processed_item
```

When and Why to Use Child Workflows

Use Cases for Child Workflows

1. **Parallel Processing:** Process multiple items concurrently
2. **Modular Design:** Break complex workflows into smaller, manageable pieces
3. **Fault Isolation:** Isolate failures to specific parts of the workflow
4. **Resource Management:** Distribute work across different workers or task queues
5. **Reusability:** Create reusable workflow components

When NOT to Use Child Workflows

- **Simple Sequential Logic:** If the workflow is simple and sequential, child workflows add unnecessary complexity
- **Tight Coupling:** If child workflows need to share a lot of state with the parent
- **Performance Overhead:** For very small tasks, the overhead of starting child workflows may not be worth it

Example: E-commerce Order Processing

```
@workflow.defn
class OrderProcessingWorkflow:
    @workflow.run
    async def run(self, order_id: str, items: list) -> dict:
        # Start child workflows for each item
        item_handles = []
        for item in items:
            handle = await workflow.start_child_workflow(
                ItemProcessingWorkflow.run,
                order_id,
                item,
                id=f"item-{order_id}-{item['id']}"
            )
            item_handles.append(handle)

        # Wait for all items to be processed
        item_results = await asyncio.gather(*item_handles)

        # Aggregate results
        total_amount = sum(result['amount'] for result in item_results)
        return {
            "order_id": order_id,
            "items": item_results,
            "total_amount": total_amount
        }
```

Spawning Child Workflows

Basic Child Workflow Spawning

```
from temporalio import workflow

@workflow.defn
class ParentWorkflow:
    @workflow.run
    async def run(self, input_data: str) -> str:
        # Start a child workflow
        child_handle = await workflow.start_child_workflow(
            ChildWorkflow.run,
            input_data,
```

```
            id="my-child-workflow"
        )

        # Wait for the child to complete
        result = await child_handle
        return f"Parent completed with child result: {result}"
```

Child Workflow Options

You can configure various options when starting child workflows:

```
child_handle = await workflow.start_child_workflow(
    ChildWorkflow.run,
    input_data,
    id="child-workflow-id",
    task_queue="child-task-queue",
    execution_timeout=timedelta(minutes=30),
    run_timeout=timedelta(minutes=25),
    task_timeout=timedelta(minutes=5),
    retry_policy=RetryPolicy(maximum_attempts=3)
)
```

Child Workflow ID Management

It's important to use unique IDs for child workflows to avoid conflicts:

```
import uuid

@workflow.defn
class ParentWorkflow:
    @workflow.run
    async def run(self, items: list) -> list:
        results = []
        for i, item in enumerate(items):
            # Generate unique ID for each child
            child_id = f"child-{workflow.info().workflow_id}-{i}-
{uuid.uuid4()}"

            child_handle = await workflow.start_child_workflow(
                ChildWorkflow.run,
                item,
                id=child_id
            )
            results.append(child_handle)

        return await asyncio.gather(*results)
```

Fan-out/Fan-in Patterns

What is Fan-out/Fan-in?

Fan-out/fan-in is a parallel processing pattern where you:

1. **Fan-out:** Start multiple child workflows concurrently
2. **Fan-in:** Wait for all child workflows to complete and aggregate their results

Basic Fan-out/Fan-in

```python
@workflow.defn
class FanOutFanInWorkflow:
    @workflow.run
    async def run(self, items: list) -> dict:
        # Fan-out: Start child workflows for all items
        child_handles = []
        for item in items:
            handle = await workflow.start_child_workflow(
                ProcessItemWorkflow.run,
                item,
                id=f"process-{item['id']}"
            )
            child_handles.append(handle)

        # Fan-in: Wait for all children to complete
        results = await asyncio.gather(*child_handles)

        # Aggregate results
        return {
            "processed_items": len(results),
            "successful_items": len([r for r in results if r['success']]),
            "failed_items": len([r for r in results if not r['success']]),
            "results": results
        }
```

Controlled Fan-out (Batch Processing)

For large datasets, you might want to process items in batches to avoid overwhelming the system:

```python
@workflow.defn
class BatchProcessingWorkflow:
    @workflow.run
    async def run(self, items: list, batch_size: int = 10) -> dict:
        all_results = []

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

```
            # Start child workflows for this batch
            batch_handles = []
            for item in batch:
                handle = await workflow.start_child_workflow(
                    ProcessItemWorkflow.run,
                    item,
                    id=f"batch-{i}-{item['id']}"
                )
                batch_handles.append(handle)

            # Wait for batch to complete
            batch_results = await asyncio.gather(*batch_handles)
            all_results.extend(batch_results)

            # Optional: Add delay between batches
            if i + batch_size < len(items):
                await workflow.sleep(1)

        return {"results": all_results}
```

Aggregating Results from Child Workflows

Simple Aggregation

```
# Wait for all children and collect results
results = await asyncio.gather(*child_handles)

# Simple aggregation
total = sum(result['value'] for result in results)
success_count = len([r for r in results if r['success']])
```

Complex Aggregation with Error Handling

```
@workflow.defn
class ComplexAggregationWorkflow:
    @workflow.run
    async def run(self, items: list) -> dict:
        child_handles = []
        for item in items:
            handle = await workflow.start_child_workflow(
                ProcessItemWorkflow.run,
                item,
                id=f"process-{item['id']}"
            )
            child_handles.append(handle)

        # Collect results with error handling
        results = []
```

```
        errors = []

        for handle in child_handles:
            try:
                result = await handle
                results.append(result)
            except Exception as e:
                errors.append({
                    "workflow_id": handle.id,
                    "error": str(e)
                })

        return {
            "successful_results": results,
            "errors": errors,
            "total_processed": len(results) + len(errors),
            "success_rate": len(results) / (len(results) + len(errors))
        }
```

Streaming Results

For long-running child workflows, you might want to process results as they complete:

```
@workflow.defn
class StreamingWorkflow:
    @workflow.run
    async def run(self, items: list) -> dict:
        # Start all child workflows
        child_handles = []
        for item in items:
            handle = await workflow.start_child_workflow(
                ProcessItemWorkflow.run,
                item,
                id=f"process-{item['id']}"
            )
            child_handles.append(handle)

        # Process results as they complete
        completed_results = []
        pending_handles = set(child_handles)

        while pending_handles:
            # Wait for any child to complete
            done, pending_handles = await asyncio.wait(
                pending_handles,
                return_when=asyncio.FIRST_COMPLETED
            )

            for handle in done:
                try:
                    result = await handle
```

```
                    completed_results.append(result)
                    workflow.logger.info(f"Child workflow {handle.id}
completed")
                except Exception as e:
                    workflow.logger.error(f"Child workflow {handle.id}
failed: {e}")

        return {"results": completed_results}
```

Advanced Error Handling and Rollback Strategies

Partial Failure Handling

```
@workflow.defn
class ResilientWorkflow:
    @workflow.run
    async def run(self, items: list) -> dict:
        # Start child workflows
        child_handles = []
        for item in items:
            handle = await workflow.start_child_workflow(
                ProcessItemWorkflow.run,
                item,
                id=f"process-{item['id']}"
            )
            child_handles.append(handle)

        # Collect results with partial failure handling
        successful_results = []
        failed_items = []

        for handle in child_handles:
            try:
                result = await handle
                successful_results.append(result)
            except Exception as e:
                failed_items.append({
                    "workflow_id": handle.id,
                    "error": str(e)
                })

        # Decide on overall workflow outcome
        if len(failed_items) > len(items) * 0.5:  # More than 50% failed
            # Rollback successful operations
            await self._rollback_successful_operations(successful_results)
            raise Exception("Too many failures, rolling back")

        return {
            "successful": successful_results,
            "failed": failed_items
```

```
        }

    async def _rollback_successful_operations(self, successful_results):
        rollback_handles = []
        for result in successful_results:
            handle = await workflow.start_child_workflow(
                RollbackWorkflow.run,
                result,
                id=f"rollback-{result['id']}"
            )
            rollback_handles.append(handle)

        await asyncio.gather(*rollback_handles, return_exceptions=True)
```

Circuit Breaker Pattern

```
@workflow.defn
class CircuitBreakerWorkflow:
    def __init__(self):
        self.failure_count = 0
        self.max_failures = 3

    @workflow.run
    async def run(self, items: list) -> dict:
        results = []

        for item in items:
            if self.failure_count >= self.max_failures:
                workflow.logger.warning("Circuit breaker open, stopping
processing")
                break

            try:
                handle = await workflow.start_child_workflow(
                    ProcessItemWorkflow.run,
                    item,
                    id=f"process-{item['id']}"
                )
                result = await handle
                results.append(result)
                self.failure_count = 0  # Reset on success
            except Exception as e:
                self.failure_count += 1
                workflow.logger.error(f"Child workflow failed: {e}")

        return {"results": results, "failure_count": self.failure_count}
```

Real-World Examples

Example 1: Data Processing Pipeline

```python
@workflow.defn
class DataProcessingPipeline:
    @workflow.run
    async def run(self, data_files: list) -> dict:
        # Stage 1: Validate files
        validation_handles = []
        for file in data_files:
            handle = await workflow.start_child_workflow(
                FileValidationWorkflow.run,
                file,
                id=f"validate-{file['name']}"
            )
            validation_handles.append(handle)

        validation_results = await asyncio.gather(*validation_handles)
        valid_files = [r for r in validation_results if r['valid']]

        # Stage 2: Process valid files
        processing_handles = []
        for file in valid_files:
            handle = await workflow.start_child_workflow(
                FileProcessingWorkflow.run,
                file,
                id=f"process-{file['name']}"
            )
            processing_handles.append(handle)

        processing_results = await asyncio.gather(*processing_handles)

        return {
            "total_files": len(data_files),
            "valid_files": len(valid_files),
            "processed_files": len(processing_results),
            "results": processing_results
        }
```

Example 2: Microservice Orchestration

```python
@workflow.defn
class MicroserviceOrchestration:
    @workflow.run
    async def run(self, user_request: dict) -> dict:
        # Start parallel microservice calls
        user_handle = await workflow.start_child_workflow(
            UserServiceWorkflow.run,
            user_request['user_id'],
            id=f"user-{user_request['user_id']}"
        )
```

```
        product_handle = await workflow.start_child_workflow(
            ProductServiceWorkflow.run,
            user_request['product_id'],
            id=f"product-{user_request['product_id']}"
        )

        inventory_handle = await workflow.start_child_workflow(
            InventoryServiceWorkflow.run,
            user_request['product_id'],
            id=f"inventory-{user_request['product_id']}"
        )

        # Wait for all services to respond
        user_data, product_data, inventory_data = await asyncio.gather(
            user_handle, product_handle, inventory_handle
        )

        # Aggregate and return combined result
        return {
            "user": user_data,
            "product": product_data,
            "inventory": inventory_data,
            "combined_result": self._combine_results(user_data,
product_data, inventory_data)
        }
```

Performance Considerations

Concurrency Limits

Be mindful of the number of concurrent child workflows:

```
@workflow.defn
class ControlledConcurrencyWorkflow:
    @workflow.run
    async def run(self, items: list, max_concurrent: int = 10) -> dict:
        semaphore = asyncio.Semaphore(max_concurrent)

        async def process_with_semaphore(item):
            async with semaphore:
                handle = await workflow.start_child_workflow(
                    ProcessItemWorkflow.run,
                    item,
                    id=f"process-{item['id']}"
                )
                return await handle

        # Process items with controlled concurrency
        tasks = [process_with_semaphore(item) for item in items]
```

```
        results = await asyncio.gather(*tasks)

        return {"results": results}
```

Resource Management

Consider the resource requirements of child workflows:

```
# Use different task queues for different types of work
child_handle = await workflow.start_child_workflow(
    HeavyComputationWorkflow.run,
    data,
    task_queue="heavy-computation-queue",  # Dedicated queue for heavy work
    id=f"heavy-{data['id']}"
)
```

Best Practices and Anti-Patterns

Best Practices

1. **Use Unique IDs:** Always provide unique IDs for child workflows
2. **Handle Failures Gracefully:** Implement proper error handling for child workflow failures
3. **Limit Concurrency:** Don't start too many child workflows simultaneously
4. **Use Appropriate Task Queues:** Route child workflows to appropriate task queues
5. **Monitor Performance:** Track the performance of child workflows

Anti-Patterns

1. **Nested Child Workflows:** Avoid deeply nested child workflows as they can be hard to debug
2. **Shared State:** Don't rely on shared state between parent and child workflows
3. **Infinite Child Workflows:** Don't create child workflows in infinite loops
4. **Ignoring Child Failures:** Always handle child workflow failures appropriately

---

Testing Child Workflows

Unit Testing

```
from unittest.mock import patch

@patch('my_module.ChildWorkflow.run')
def test_parent_workflow_spawns_children(mock_child):
    mock_child.return_value = {"result": "success"}

    # Test that parent workflow starts child workflows
    result = run_workflow(ParentWorkflow, ["item1", "item2"])
```

```
    assert mock_child.call_count == 2
    assert result["results"] == [{"result": "success"}, {"result":
"success"}]
```

Integration Testing

Test child workflows with a real Temporal server:

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

    # Start parent workflow
    handle = await client.start_workflow(
        ParentWorkflow.run,
        ["item1", "item2"],
        id="test-parent"
    )

    # Wait for completion
    result = await handle.result()

    # Verify results
    assert len(result["results"]) == 2
```

Troubleshooting Common Issues

Common Issues and Solutions

1. **Child Workflow Not Starting:**
   - Check that the child workflow is registered with the worker
   - Verify the task queue configuration
   - Ensure the child workflow ID is unique
2. **Child Workflow Hanging:**
   - Check for infinite loops in child workflow logic
   - Verify timeout configurations
   - Look for blocking operations
3. **Memory Issues with Many Child Workflows:**
   - Implement batching to limit concurrent child workflows
   - Use different task queues to distribute load
   - Monitor worker memory usage
4. **Child Workflow Failures Not Handled:**
   - Always wrap child workflow calls in try/except blocks

    - Implement proper error handling and logging
    - Consider using circuit breaker patterns

Summary

Today, we explored child workflows and parallel processing patterns in Temporal:

- Understanding when and why to use child workflows
- Spawning and managing child workflows from parent workflows
- Implementing fan-out/fan-in patterns for parallel processing
- Aggregating results efficiently from multiple child workflows
- Advanced error handling and rollback strategies
- Performance considerations and best practices

Key Takeaways:

- Child workflows provide modularity and fault isolation
- Fan-out/fan-in patterns enable efficient parallel processing
- Always handle child workflow failures gracefully
- Monitor performance and resource usage
- Use appropriate task queues and concurrency limits

Tomorrow, we'll explore workflow versioning and long-running flows, essential for maintaining and upgrading workflows in production.