Infrastructure-Aware Task Execution / Context Propagation

Airflow Improvement Proposal

Status: Draft
Related Proposal: Resumable Operators for Disruption Readiness
Discussion Thread: [DISCUSS] Infrastructure-Aware Task Execution and Resumable Operators - Proposals for Reliability-Apache Mail Archives
Vote Thread: TBD
Authors: Stefan Wang (1fannnw@gmail.com)
Created: November 11, 2024
Target Version: Airflow 3.x

Summary

This document proposes enhancing Airflow's task execution reliability by enabling infrastructure-aware decisions during failures and terminations. It introduces **Execution Context Propagation** and **Infrastructure Failure Auto-Retry** to help Airflow distinguish between infrastructure issues (worker crashes, pod evictions) and application errors (user code bugs), enabling smarter retry budgets and better operational clarity.

Key Benefits:

- Platform teams can accurately attribute failures (infrastructure vs application)
- Users' retry budgets are protected from infrastructure disruptions
- Operators can make intelligent cleanup decisions (preserve vs cancel remote jobs)
- Clear observability through listener hooks with rich failure context

Motivation

Current Behavior and Limitations

Problem 1: No Observability Context for Root Cause Analysis
When DAG runs fail, listener hooks receive only the exception object with no structured context:

```
@hookimpl
def on_task_instance_failed(previous_state, task_instance, error):
    #  'error' is just the exception
    #  No category (infrastructure vs application)
```

```
# No source (executor, scheduler, worker)
# No reason (pod eviction vs timeout vs OOM)
```

Platform teams must manually traverse logs and task states to determine root causes.

Problem 2: Infrastructure Failures Consume User Retry Budgets

Users configure retries for application issues, but infrastructure failures silently consume them:

```
@task(retries=3)  # "I want 3 retries for data processing job issues"
def process_data():
    cleaned = clean_data(raw_data)
    if not validate(cleaned):
        raise DataProcessingError("Invalid data")
```

What actually happens:

- Try 1: DNS failure during worker init → Infrastructure
- Try 2: K8s pod evicted → Infrastructure
- Try 3: DataProcessingError in user code → Application
- Result: Failed permanently → User: *"I only got 1 real retry!"*

Why is it needed?

Scenario 1: The Platform Team's Operational Dashboard

User: Alex, a platform engineer monitoring 1000s of DAGs

Current Experience:

- DAG fails, listener hook gets called
- No context at all about *why* it failed
- Must manually check logs, examine task states, infer failure type
- Cannot reliably route alerts (infrastructure issues should page platform team, application issues should notify data team)

With This Proposal:

```
@hookimpl
def on_task_instance_failed(previous_state, task_instance, error, execution_context):
    if execution_context.category == StateChangeCategory.INFRASTRUCTURE:
        metrics.incr("task.failed.infrastructure",
                     tags={"reason": execution_context.reason})
    else:
        metrics.incr("task.failed.application")
```

Result: Clear attribution, accurate metrics, proper alert routing.

Scenario 2: The Frustrated Data Scientist

User: Jordan, carefully configures retry budgets for transient data issues

Current Experience:

- Configures retries=5 for API rate limits and data quality checks
- Infrastructure issues (DNS failures, pod evictions) consume 3-4 retries
- Actual application error exhausts remaining retries
- Confusion: *"Why did my task fail after 5 retries when I only saw 1 data quality error?"*

With This Proposal:

```
# Platform config
infrastructure_retry_budget = 5

# User config (unchanged)
@task(retries=3)
def process_data():
    ...
```

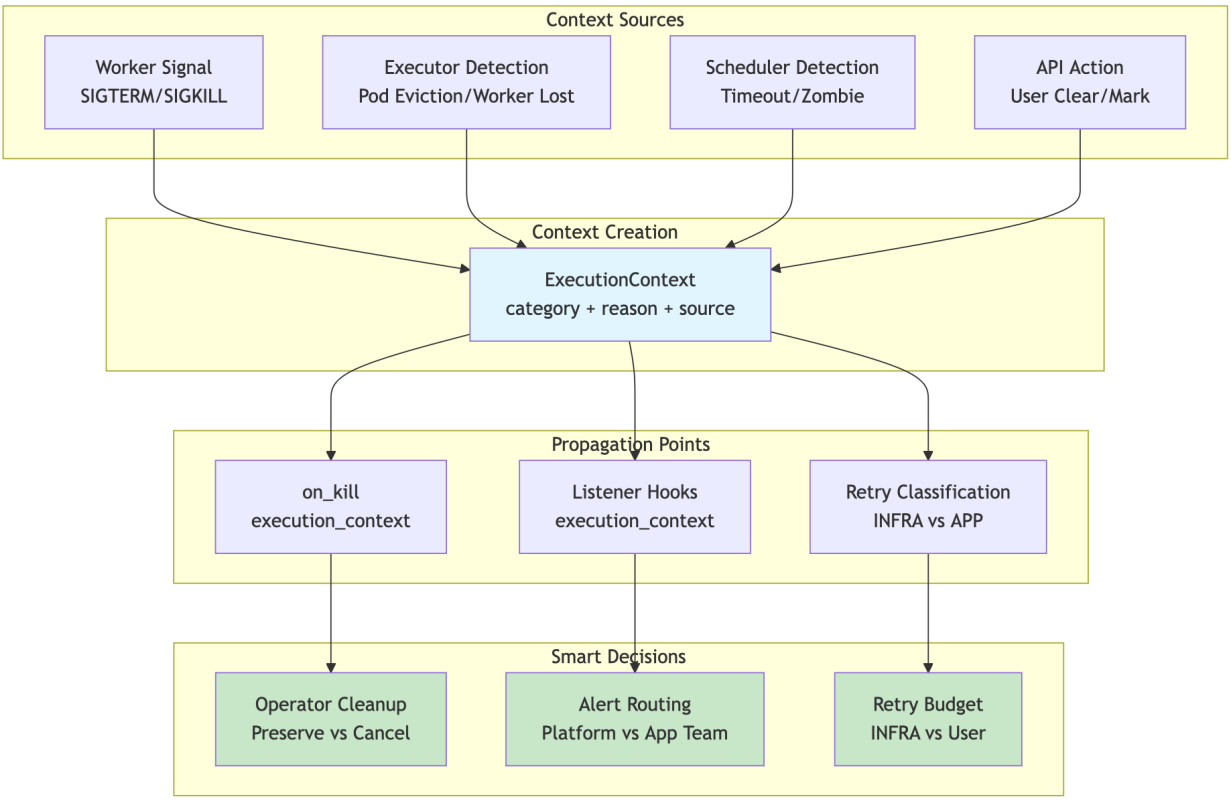
Result:

- Infrastructure gets 5 automatic retries (invisible to user)
- User's 3 retries protected for actual application issues
- Clean UX: "Task Succeeded - Attempt 1/3" (infrastructure retries hidden)

What change do you propose to make?

Architecture Overview

The following diagram shows how Execution Context flows through the system:



Part 1: Execution Context Propagation

Core Concept

Provide rich, actionable context for all task state changes. Context flows from the source of truth (executor signals, scheduler decisions, API actions) rather than being inferred post-hoc.

Data Structure

```
@dataclass
class ExecutionContext:
    """Context for state transitions."""

    category: StateChangeCategory
    # - INFRASTRUCTURE: worker crash, pod eviction, DB connection loss
    # - APPLICATION: user code exception, data validation error
    # - TIMEOUT: execution timeout exceeded
    # - USER_ACTION: manual clear/mark via UI/API

    reason: StateChangeReason
    # - WORKER_TERMINATION (SIGTERM/SIGKILL)
```

```
    # - WORKER_LOST (heartbeat timeout)
    # - RESOURCE_EXHAUSTION (OOM, disk full)
    # - DB_CONNECTION_ERROR (transient DNS/network)
    # - EXECUTION_TIMEOUT (task timeout)
    # - MANUAL_CLEAR (user action)

    source: ContextSource
    # - WORKER: signal received in task process
    # - EXECUTOR: executor detected issue
    # - SCHEDULER: timeout/zombie detection
    # - API: user action via UI/API
    metadata: Dict[str, Any]  # Executor-specific details
```

Propagation Points

1. Enhanced on_kill() signature:

```
class BaseOperator:
    def on_kill(self, execution_context: ExecutionContext | None = None) -> None:
        """
        Called when the task is terminated.
        Args:
            execution_context: Rich context about WHY termination occurred.
                Available in Airflow 3.x+. None for backward compatibility.
        """
        pass  # Operators override this
```

2. Enhanced listener hooks:

```
@hookspec
def on_task_instance_failed(
    previous_state: TaskInstanceState | None,
    task_instance: TaskInstance,
    error: None | str | BaseException,
    execution_context: ExecutionContext | None = None,  # NEW
):
    """Execute when task state changes to FAIL."""
```

3. Context creation at source:

```
# TI/Worker receives signal
def signal_handler(signum, frame):
    context = ExecutionContext(
        category=StateChangeCategory.INFRASTRUCTURE,
```

```
        reason=StateChangeReason.WORKER_TERMINATION,
        source=ContextSource.WORKER,
        metadata={'signal': signum}
    )
    task.on_kill(context)

# Executor detects pod eviction (K8s)
if pod.status.reason == 'Evicted':
    context = ExecutionContext(
        category=StateChangeCategory.INFRASTRUCTURE,
        reason=StateChangeReason.WORKER_LOST,
        source=ContextSource.EXECUTOR,
        metadata={'pod_reason': 'Evicted', 'node': pod.spec.node_name}
    )
    self.fail_task(ti, context)
```

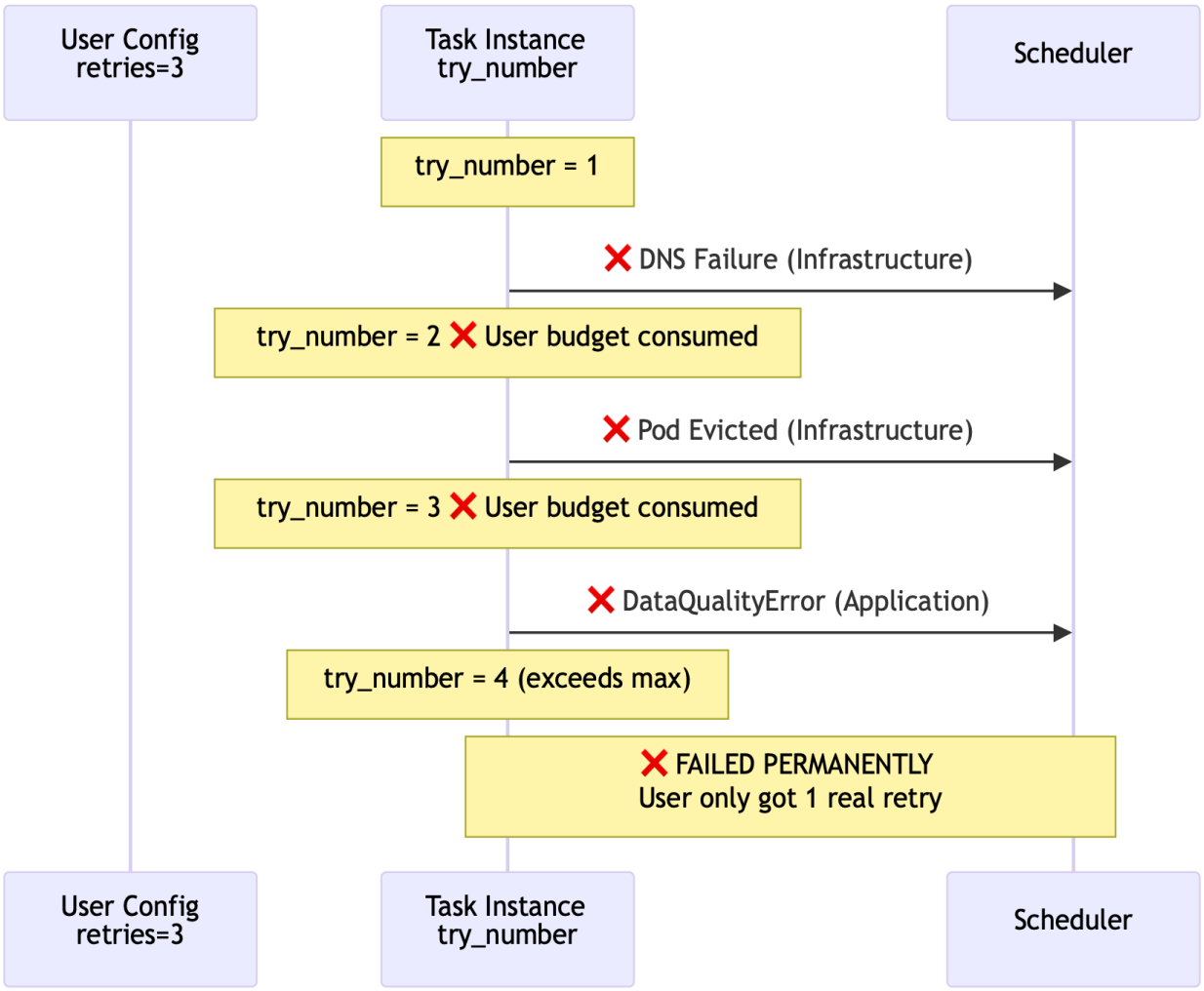
Part 2: Infrastructure Failure Auto-Retry

Core Concept

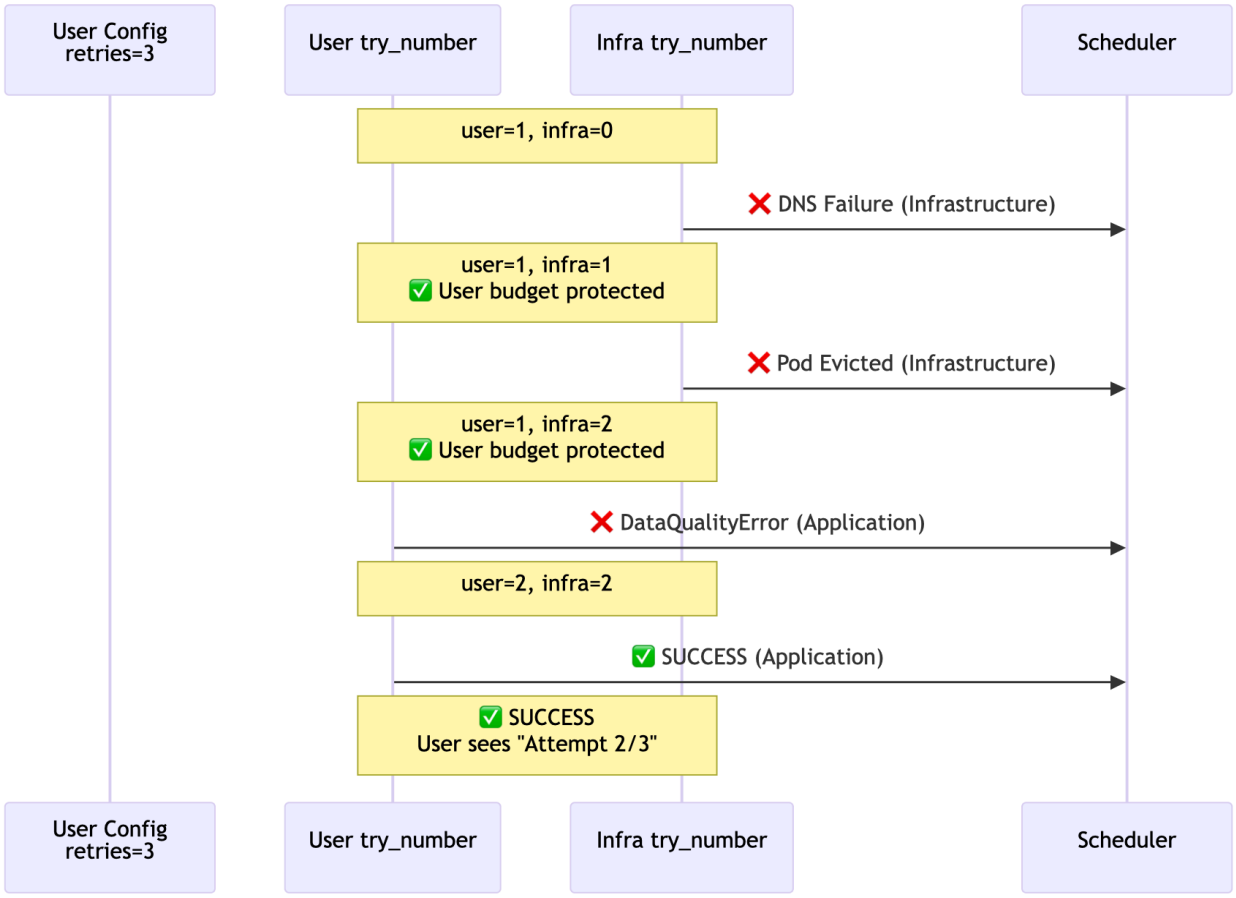
Separate retry budgets: platform-managed retries for infrastructure failures, user-configured retries for application errors.

Retry Flow Comparison

Current Behavior (Single Budget):



Proposed Behavior (Separate Budgets):



Implementation

Retry Classification Logic:

```
def classify_for_retry(context: ExecutionContext) -> RetryType:
    """Determine which retry budget to use."""
    if context.category == StateChangeCategory.INFRASTRUCTURE:
        # Retryable infrastructure issues
        if context.reason in [
            StateChangeReason.WORKER_TERMINATION,
            StateChangeReason.WORKER_LOST,
            StateChangeReason.RESOURCE_EXHAUSTION,
            StateChangeReason.DB_CONNECTION_ERROR,
        ]:
            return RetryType.INFRASTRUCTURE

    # Application failures or non-retryable infrastructure
    return RetryType.APPLICATION
```

Budget Consumption:

```
# Infrastructure failure
if retry_type == RetryType.INFRASTRUCTURE:
   ti.infrastructure_try_number += 1
   ti.try_number -= 1  # Auto-decrement to keep user view unchanged

# Application failure
else:
   ti.try_number += 1  # Normal behavior
```

User Experience

Current:

Logs: Try 1, Try 2, Try 3 (all visible, confusing)

UI: "Task Failed - Attempt 3/3"

User: "Why did it fail? I saw 1 data error but used all 3 retries!"

Proposed:

User Logs: Try 1 (clean, only app retries shown)

UI: "Task Succeeded - Attempt 1/3"

Platform Logs: "Infrastructure retry 2/5 succeeded" (separate tracking)

User: "My retry budget works as expected!"

Configuration

```
# airflow.cfg - Platform defaults
[scheduler]
infrastructure_retry_budget = 5
infrastructure_retry_delay = 10
# Per-executor tuning
[kubernetes_executor]
infrastructure_retry_budget = 7  # More volatile
```

```
# Task-level override (optional)
@task(
   retries=3,  # User's application retries
   infrastructure_retry_enabled=True,  # Default: from config
   infrastructure_retry_limit=7,  # Override platform default
)
def process_data():
   pass
```

What problem does it solve?

1. **Smart Cleanup Decisions**: Operators can distinguish temporary infrastructure issues from timeouts, preserving expensive remote jobs when appropriate
2. **Protected User Retry Budgets**: Infrastructure failures don't consume user-configured retries, eliminating user confusion
3. **Clear Operational Attribution**: Platform teams get accurate metrics and can route alerts to the right teams (platform vs application)
4. **Better Reliability**: Automatic infrastructure retries improve overall system reliability without user intervention

Are there any downsides to this change?

Minimal:

- Additional database column: infrastructure_try_number (INTEGER)
- Additional database column: last_failure_context (JSONB)
- Slight increase in metadata passed between components

Mitigations:

- All changes are backward compatible (new parameters optional, default to None)
- Existing operators/plugins work unchanged
- Indexes added for performance

Which users are affected by the change?

Positively Affected:

- **All Users**: Benefit from infrastructure auto-retry (transparent)
- **DAG Authors**: Can implement smarter operator cleanup logic
- **Platform Teams**: Get rich observability context for monitoring
- **Provider Maintainers**: Can build more resilient operators (Databricks, EMR, Snowflake, etc.)

Not Affected:

- Users who don't implement execution_context logic (backward compatible)
- Existing listener plugins (context parameter is optional)

How are users affected by the change?

Database Migration

```
ALTER TABLE task_instance
  ADD COLUMN infrastructure_try_number INTEGER DEFAULT 0 NOT NULL,
  ADD COLUMN last_failure_context JSONB;

CREATE INDEX idx_ti_infrastructure_retries
  ON task_instance(infrastructure_try_number)
  WHERE infrastructure_try_number > 0;
```

Code Changes (Opt-in)

Operators (optional enhancement):

```
# Before: no context
def on_kill(self):
  self.cancel_job(self.job_id)

# After: context-aware (opt-in)
def on_kill(self, execution_context=None):
  if execution_context and execution_context.category == INFRASTRUCTURE:
    self.preserve_job(self.job_id)
  else:
    self.cancel_job(self.job_id)
```

Listener Plugins (optional enhancement):

```
# Before: just error
def on_task_instance_failed(previous_state, task_instance, error):
  log_failure(error)

# After: rich context (opt-in)
def on_task_instance_failed(previous_state, task_instance, error, execution_context=None):
  if execution_context:
    route_alert_by_category(execution_context.category)
  log_failure(error)
```

What is the level of migration effort?

Zero Breaking Changes

- Default behavior unchanged (infrastructure retry opt-in via config)
- All new parameters optional with safe defaults
- Existing operators work without modification
- Existing listener plugins work without modification

Gradual Adoption Path

Phase 1: Foundation (Airflow 3.x)
- Add ExecutionContext model
- Add optional context parameters to interfaces
- Default: infrastructure retry disabled

Phase 2: Adoption (Airflow 3.x+1)
- Providers update operators to use context
- Documentation with migration examples
- Users enable infrastructure retry per-DAG or per-pool

Phase 3: Default Enabled (Airflow 3.x+2)
- Infrastructure retry enabled by default
- Monitor and tune budgets
- Full production readiness

What defines this AIP as "done"?

1. ExecutionContext dataclass implemented
2. on_kill(execution_context) signature updated in BaseOperator
3. Listener hooks signatures updated with execution_context parameter
4. Signal handlers create and propagate context
5. Executors create context from infrastructure signals
6. Infrastructure retry classification logic implemented
7. Separate retry budget tracking (infrastructure_try_number)
8. Database migrations added
9. Configuration options added to airflow.cfg
10. Documentation updated with examples
11. Metrics added (task.failed.infrastructure, task.failed.application)
12. Tests added for all retry scenarios

Appendix

Appendix A: Context Detection Mechanisms

Kubernetes Executor Example

```
def create_context_from_pod(pod: V1Pod) -> ExecutionContext:
    """Leverage existing rich signals from K8s."""

    if pod.status.reason == 'Evicted':
        return ExecutionContext(
            category=StateChangeCategory.INFRASTRUCTURE,
            reason=StateChangeReason.RESOURCE_EXHAUSTION,
            metadata={
                'pod_reason': pod.status.reason,
                'node': pod.spec.node_name,
                'container_reason': pod.status.container_statuses[0].state.terminated.reason
            }
        )

    if pod.status.phase == 'Failed':
        exit_code = pod.status.container_statuses[0].state.terminated.exit_code
        if exit_code == 137:  # SIGKILL
            return ExecutionContext(
                category=StateChangeCategory.INFRASTRUCTURE,
                reason=StateChangeReason.WORKER_TERMINATION,
                metadata={'exit_code': 137, 'signal': 'SIGKILL'}
            )
```

Transient Database Errors (All Executors)

```
def is_retryable_infrastructure_error(exception: BaseException) -> bool:
    """Detect transient DB infrastructure errors."""
    if isinstance(exception, (OperationalError, DBAPIError)):
        error_code = extract_db_error_code(exception)
        # MySQL: 2005 (DNS failure), 2013 (lost connection)
        # PostgreSQL: 08006 (connection failure), 08000 (connection exception)
        if error_code in RETRYABLE_DB_ERROR_CODES:
            return True
    return False
```

Appendix B: Metrics

New Metrics (OTel-first):

```
# Task failure metrics tagged by category
Stats.incr(
    "task_instance.failed",
    tags={
        "dag_id": ti.dag_id,
        "task_id": ti.task_id,
        "category": execution_context.category.value,  # infrastructure/application
        "reason": execution_context.reason.value,
    }
)

# Infrastructure retry metrics
Stats.incr(
    "task_instance.infrastructure_retry",
    tags={
        "dag_id": ti.dag_id,
        "attempt": ti.infrastructure_try_number,
    }
)
```

Appendix C: Open Questions for Community

1. **Naming**: Is INFRASTRUCTURE vs APPLICATION clear? Alternative: PLATFORM vs USER?
2. **Default Behavior**: Should infrastructure retry be opt-in or opt-out?
3. **Budget Scope**: Should defaults be per-executor, per-pool, or global?
4. **Heuristic Detection**: For signals without rich context, should we use heuristics (e.g., task still RUNNING when SIGTERM = likely infrastructure)?

References

Airflow Source

- TaskDeferred exception
- K8s FailureDetails
- Current on_kill() signature

External References

- Flyte Error Classification
- Flyte Retry Budgets