Data Pipeline / ETL System: Design Document

Overview

This system builds a scalable ETL pipeline framework that orchestrates data movement, transformation, and validation across heterogeneous sources. The key architectural challenge is managing complex task dependencies, handling failures gracefully, and ensuring data consistency while maintaining high throughput and observability.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

> **Milestone(s):** Foundation for all milestones - establishes the architectural context and requirements that guide the entire system design.

Mental Model: Factory Assembly Line

Think of an ETL pipeline system as orchestrating a massive **factory assembly line** where raw materials flow through multiple processing stations to create finished products. Just as Henry Ford revolutionized manufacturing by breaking down car assembly into discrete, coordinated steps, modern data processing requires breaking down complex data workflows into manageable, interdependent tasks.

In our factory analogy, **raw data sources** are like suppliers delivering different types of materials - some arrive by truck (batch files), others through conveyor belts (streaming APIs), and still others from warehouse inventory (databases). Each **transformation step** is a specialized workstation with specific tools and workers trained for particular operations - cutting, welding, painting, or quality inspection. The **final destinations** are shipping docks where finished products are loaded onto trucks bound for different customers.

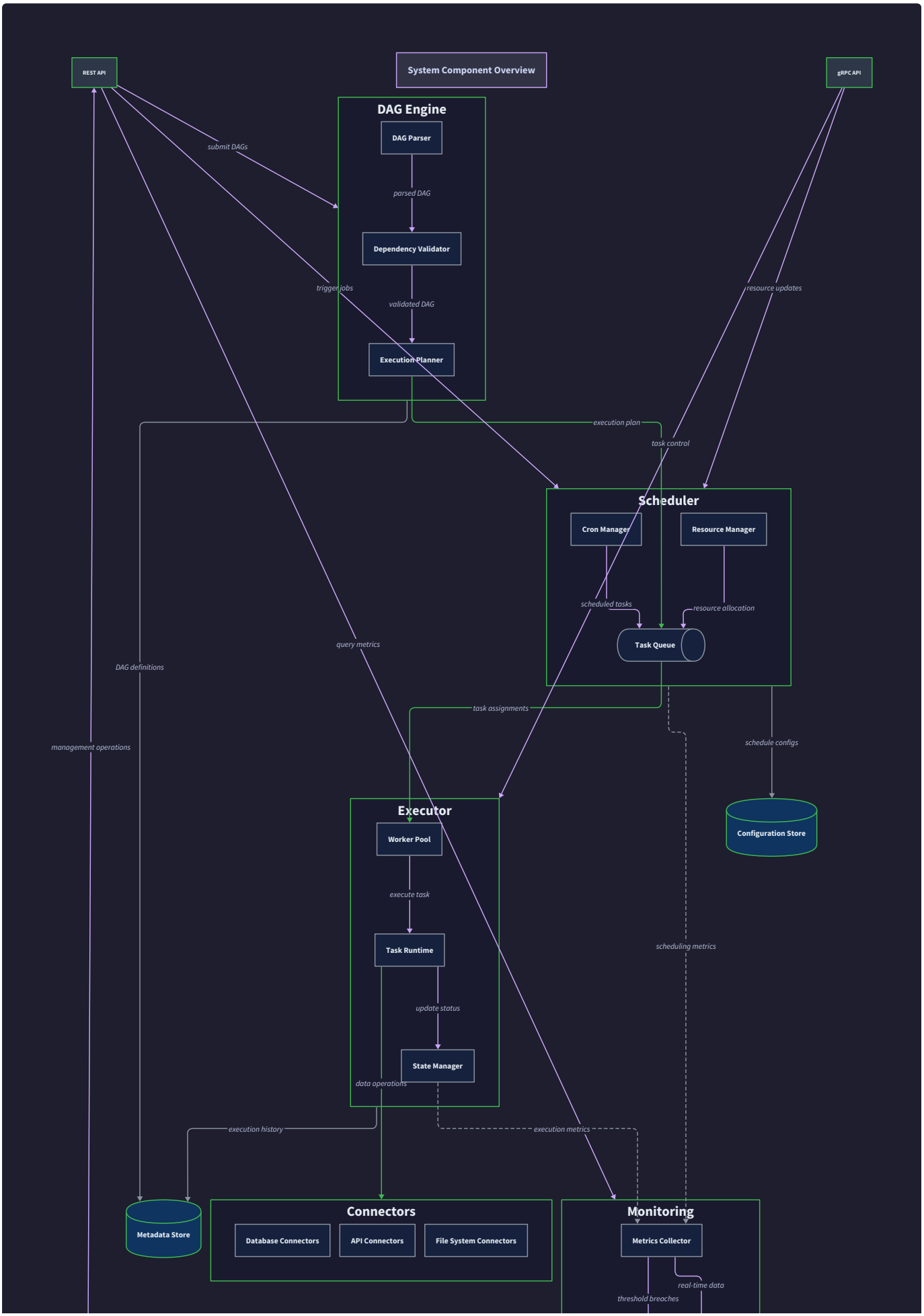
The critical insight from manufacturing applies directly to data processing: **dependencies matter immensely**. You cannot paint a car door before it's been cut and shaped. You cannot install wheels before the chassis is assembled. Similarly, you cannot aggregate sales data before cleaning customer records, and you cannot generate reports before all source systems have been synchronized. This dependency management becomes the central orchestration challenge.

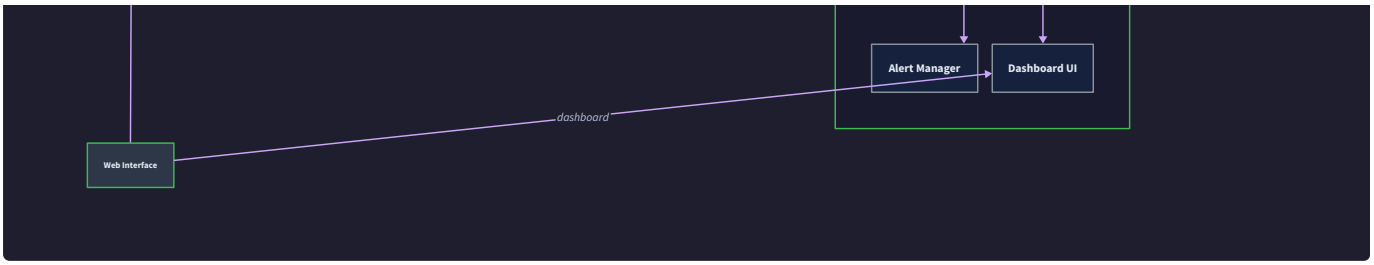
Just as a factory needs a **production control system** to track work orders, monitor station capacity, handle equipment failures, and ensure quality standards, our ETL system needs sophisticated orchestration to manage task scheduling, resource allocation, failure recovery, and data validation. The complexity multiplies when you consider that our "factory" operates 24/7 across distributed systems, with "materials" arriving from dozens of sources on different schedules, and "customers" expecting deliveries with strict SLA requirements.

The assembly line analogy also reveals why simple script-based approaches fail at scale. A small workshop can operate with informal coordination - the craftsperson handles each order from start to finish. But when you need to process terabytes of data daily with sub-hour latency requirements, you need the equivalent of Toyota's production system: standardized processes, quality gates, just-in-time delivery, and sophisticated error recovery mechanisms.

Existing ETL Solutions

The ETL landscape today offers several mature platforms, each optimizing for different aspects of the assembly line metaphor. Understanding their design philosophies and trade-offs illuminates the architectural decisions our system must make.


System Component Overview
REST API
gRPC API
DAG Engine
DAG Parser
Dependency Validator
Execution Planner
submit DAGs
parsed DAG
validated DAG
trigger jobs
resource updates
execution plan
task control
Scheduler
Cron Manager
Resource Manager
scheduled tasks
resource allocation
Task Queue
query metrics
DAG definitions
task assignments
schedule configs
management operations
Executor
Worker Pool
execute task
Task Runtime
update status
State Manager
data operations
Configuration Store
scheduling metrics
execution history
execution metrics
Metadata Store
Connectors
Database Connectors
API Connectors
File System Connectors
Monitoring
Metrics Collector
real-time data
threshold breaches



Apache Airflow: The Enterprise Standard

Apache Airflow dominates enterprise ETL orchestration by treating pipelines as **Directed Acyclic Graphs (DAGs) defined in Python code**. Airflow's strength lies in its comprehensive ecosystem and battle-tested scalability - it orchestrates workflows for companies processing petabytes daily across thousands of tasks.

Aspect	Strength	Limitation	Impact on Design
DAG Definition	Python-native, familiar to data engineers	Code-heavy configuration, version control complexity	Need balance between code flexibility and declarative simplicity
Task Execution	Robust parallelization, resource management	Heavy infrastructure requirements (Redis, Postgres)	Must consider deployment complexity vs. feature richness
Failure Handling	Comprehensive retry policies, alerting integration	Complex debugging when failures cascade through dependencies	Error handling must be intuitive and provide clear diagnosis
Extensibility	Rich plugin ecosystem, custom operators	Steep learning curve, opinionated architecture	Plugin system should be simple but powerful

Airflow's **scheduler-executor-webserver architecture** separates concerns cleanly but requires significant operational overhead. The scheduler parses DAGs, determines task readiness, and queues work. Executors (LocalExecutor, CeleryExecutor, KubernetesExecutor) handle the actual task execution with different scalability and isolation characteristics. The webserver provides monitoring and manual intervention capabilities.

A critical Airflow insight is that **task definitions must be idempotent** because the scheduler may retry tasks multiple times due to infrastructure failures. This idempotency requirement permeates every aspect of pipeline design and significantly influences our transformation engine architecture.

Dagster: The Developer Experience Focus

Dagster represents a newer generation of orchestration tools prioritizing **developer experience and data quality**. While Airflow focuses on workflow orchestration, Dagster emphasizes the data itself - assets, lineage, and validation.

Design Philosophy	Dagster Approach	Traditional ETL Approach	Architectural Implication
Mental Model	Data assets with materialization logic	Tasks that happen to process data	Our system should be data-centric, not just task-centric
Testing	Built-in unit testing for individual ops	Integration testing in staging environments	Need comprehensive testing framework from the start
Type System	Strong typing with runtime validation	Schema-on-read with runtime failures	Type safety should be configurable, not mandatory
Observability	Asset lineage and data quality metrics built-in	Monitoring added as afterthought	Lineage tracking should be first-class, not bolted on

Dagster's **software-defined assets** concept treats data tables, files, and ML models as first-class citizens with explicit dependencies. Instead of thinking "run this SQL query after that API call completes," you think "materialize the customer_summary table when

clean_customer_data and recent_orders assets are available." This abstraction level significantly improves reasoning about complex pipelines.

The Dagster **op (operation) and job** model separates reusable logic (ops) from specific pipeline definitions (jobs). This separation enables better testing - you can unit test individual ops with mock data, then compose them into jobs for integration testing. Our system should adopt similar composability principles.

Custom Solutions: The Build vs. Buy Decision

Many organizations build custom ETL solutions when existing tools don't match their specific requirements or constraints. Common drivers for custom development include:

Requirement	Why Existing Tools Fall Short	Custom Solution Advantage	Trade-off
Extreme Performance	General-purpose tools have abstraction overhead	Optimize for specific data patterns and volumes	Development and maintenance cost vs. performance gain
Legacy Integration	Existing tools don't support proprietary protocols	Direct integration with legacy systems	Technical debt vs. operational efficiency
Regulatory Compliance	Standard tools may not meet audit requirements	Full control over security and logging	Compliance assurance vs. feature velocity
Cost Optimization	Commercial tools expensive at scale	No licensing fees, cloud-optimized architecture	Development investment vs. operational savings

Custom solutions often excel in narrow domains but struggle with the **feature breadth** that mature platforms provide. A custom solution might perfectly handle your specific data transformation needs but lack comprehensive monitoring, alerting, user management, and operational tooling that enterprises require.

> **The Build vs. Buy Decision Framework**: Choose custom development when the performance, integration, or cost advantages outweigh the opportunity cost of not investing in business logic development. The hidden costs of custom platforms include ongoing maintenance, feature development, security updates, and knowledge transfer as teams change.

Core Technical Challenges

Building a production-grade ETL system requires solving fundamental distributed systems challenges while maintaining the developer experience that makes complex data workflows manageable. These challenges are not independent - decisions in one area create constraints and opportunities in others.

Dependency Management: The Coordination Problem

Managing task dependencies in distributed systems involves more complexity than simple precedence constraints. In our assembly line analogy, this is equivalent to coordinating multiple production lines with shared resources, variable processing times, and occasional equipment failures.

The DAG Parsing Challenge: Unlike traditional job schedulers that work with simple precedence lists, ETL systems must parse potentially complex dependency graphs from various sources (YAML files, Python code, database configurations) and validate them for correctness before execution begins.

Parsing Requirement	Technical Challenge	Solution Approach	Failure Mode
Cycle Detection	Must identify circular dependencies in potentially large graphs	Implement depth-first search with back-edge detection	False positives when parsing incremental updates
Dynamic Dependencies	Task dependencies may depend on runtime data or external conditions	Support conditional dependencies with runtime resolution	Deadlocks when conditions create cycles
Cross-Pipeline Dependencies	Tasks in different pipelines may depend on each other	Global dependency resolution with pipeline isolation	Cascading failures across pipeline boundaries
Parameterized Dependencies	Template-based task definitions with variable substitution	Parse dependencies after parameter resolution	Invalid graphs when parameters create cycles

The Topological Ordering Problem: Once dependencies are validated, the system must determine execution order while maximizing parallelism. This is a classic topological sort problem, but ETL systems add complications:

1. **Resource Constraints**: Not all ready tasks can execute simultaneously due to memory, CPU, or external system limits
2. **Priority Scheduling**: Critical path tasks should execute before non-critical tasks when resources are limited
3. **Dynamic Rescheduling**: Task failures require recomputing execution order for remaining tasks
4. **Partial Pipeline Execution**: Users often want to run subsets of pipelines, requiring dependency subgraph extraction

> **Design Insight**: The dependency resolution system is the architectural foundation that determines system scalability. A naive implementation that recomputes the entire topological order after each task completion will become the bottleneck as pipeline complexity grows. The system must maintain incremental data structures that support efficient updates.

Distributed State Consistency: In a multi-node deployment, task scheduling decisions must be coordinated across schedulers to prevent duplicate execution while maintaining high availability.

Consistency Model	Coordination Mechanism	Pros	Cons
Single Leader	One scheduler node makes all decisions	Simple, consistent, no conflicts	Single point of failure, scalability bottleneck
Consensus-Based	Raft/Paxos for scheduling decisions	High availability, consistency guarantees	Complex implementation, network partition sensitivity
Eventually Consistent	Distributed schedulers with conflict resolution	High availability, partition tolerance	Potential duplicate execution, complex recovery
External Coordination	Use existing systems (etcd, Zookeeper)	Proven reliability, operational familiarity	External dependency, network latency overhead

Failure Recovery: Building Resilient Data Workflows

Data processing failures occur at multiple levels - infrastructure, network, application logic, and data quality. Unlike web applications where failed requests can be retried immediately, ETL failures often require sophisticated recovery strategies because of data consistency requirements and resource costs.

Failure Classification and Response Strategies: Different failure types require different recovery approaches, and the system must automatically classify failures to apply appropriate remediation.

Failure Type	Detection Method	Recovery Strategy	Prevention Approach
Transient Network	Connection timeout, DNS resolution failure	Exponential backoff retry	Circuit breaker pattern, connection pooling
Resource Exhaustion	Out of memory, disk full, CPU throttling	Queue task for later execution	Resource monitoring, preemptive scaling
Data Quality Issues	Schema validation failure, constraint violations	Skip bad records with logging	Data profiling, upstream validation
Logic Errors	Application exceptions, assertion failures	Human intervention required	Comprehensive testing, gradual rollouts
Upstream System Unavailable	API rate limiting, database maintenance	Defer execution, alert operators	SLA monitoring, alternative data sources

The Partial Failure Problem: ETL tasks often process large datasets where partial completion is valuable. A task that successfully processes 80% of records before encountering corrupt data should preserve its progress rather than starting from scratch.

Consider a task that extracts customer records from a REST API with pagination. The task successfully processes pages 1-8 but encounters a malformed response on page 9. The recovery options are:

1. **Full Restart**: Discard all progress and restart from page 1 - simple but wasteful
2. **Checkpoint Resume**: Save pagination state after each page and resume from page 9 - complex but efficient
3. **Partial Success**: Mark pages 1-8 as complete, fail page 9, and let dependent tasks proceed with available data - requires sophisticated dependency management

> **Architectural Decision**: The system must support **checkpoint-based recovery** as a first-class concept. Tasks should be able to save intermediate state that enables resumption after failures. This requires careful API design to make checkpointing efficient without overwhelming task implementations with persistence concerns.

Cascading Failure Prevention: When upstream tasks fail, downstream tasks must decide whether to skip execution, execute with partial data, or wait for upstream recovery. This decision depends on business requirements and data freshness constraints.

Downstream Strategy	When to Use	Implementation Complexity	Business Impact
Fail Fast	Critical data dependencies, regulatory requirements	Low - propagate failure status	High - entire pipeline stops
Partial Execution	Analytics workloads, non-critical reporting	Medium - handle missing data gracefully	Medium - reduced data quality
Wait and Retry	Near real-time requirements, SLA commitments	High - timeout management, resource allocation	Low - maintains data quality
Use Cached Data	Dashboard updates, periodic reports	Medium - cache invalidation logic	Variable - depends on data staleness tolerance

Data Consistency: Managing State Across Distributed Operations

ETL operations must maintain data consistency across multiple systems while providing reasonable performance. Unlike OLTP databases that handle small, frequent transactions, ETL systems deal with large bulk operations that may run for hours and interact with systems that don't support transactions.

The Two-Phase Load Problem: Many ETL workflows follow an "extract-transform-load" pattern where the final load step must be atomic despite potentially loading data to multiple destination systems. Consider a pipeline that loads both a data warehouse and a search index - both destinations must be updated or neither should be updated.

Traditional database transactions don't apply because:

1. **Cross-System Boundaries**: Destinations may be different database types, file systems, or external APIs
2. **Long-Running Operations**: Bulk loads may take hours, exceeding reasonable transaction timeouts
3. **Resource Constraints**: Holding locks during large operations blocks other concurrent workflows

Idempotency Requirements: Since the system may retry failed operations, all transformations and loads must produce the same result when executed multiple times. This requirement significantly influences API design and data modeling.

Operation Type	Idempotency Challenge	Solution Pattern	Implementation Notes
Incremental Loads	Duplicate records on retry	Upsert with deterministic keys	Requires unique key identification
Aggregations	Double-counting on retry	Checkpoint aggregation state	Complex with streaming data
File Operations	Partial writes on failure	Atomic rename operations	Requires staging area
API Calls	Side effects on retry	Idempotency tokens	Requires upstream system support

Schema Evolution Management: Production ETL systems must handle schema changes in source systems without breaking downstream consumers. This is particularly challenging because schema changes often occur without coordination between teams.

The schema evolution problem manifests in several ways:

- **Additive Changes**: New columns in source tables may break transformations that use `SELECT *`
- **Breaking Changes**: Removed or renamed columns cause immediate pipeline failures
- **Type Changes**: Column type modifications may cause silent data truncation or conversion errors
- **Constraint Changes**: New validation rules may reject previously acceptable data

> **Design Principle**: The system should implement **schema versioning** with backward compatibility policies. Each data asset should declare its expected schema version, and the system should validate compatibility before execution and provide automatic migration paths for compatible changes.

Scalability Challenges: Performance and Resource Management

ETL systems face unique scalability challenges because workloads are often **bursty, resource-intensive, and deadline-driven**. Unlike web applications with relatively predictable load patterns, ETL systems may need to scale from processing megabytes during quiet periods to terabytes during month-end batch jobs.

Resource Allocation and Queueing: The system must efficiently allocate compute, memory, and I/O resources across concurrent tasks while respecting priority constraints and resource limits.

Resource Type	Constraint Pattern	Scaling Strategy	Monitoring Approach
CPU	Compute-intensive transformations	Horizontal scaling, distributed processing	CPU utilization per task type
Memory	Large dataset operations, caching	Memory limits per task, spill-to-disk	Memory high-water marks
I/O	Database connections, file system throughput	Connection pooling, async operations	Queue depths, latency percentiles
Network	Cross-region data movement	Compression, regional caching	Bandwidth utilization, transfer costs

The Thundering Herd Problem: When upstream data becomes available, multiple downstream tasks may become eligible for execution simultaneously. Without careful resource management, this can overwhelm the execution infrastructure and cause cascading failures.

Consider a nightly batch job that loads customer transaction data. When the load completes, dozens of downstream analytics tasks become ready - customer segmentation, fraud detection, reporting aggregations, and ML feature generation. If all these tasks start immediately, they may:

- Overwhelm the source database with concurrent queries
- Exhaust memory by loading overlapping datasets
- Saturate network bandwidth and slow each other down
- Cause resource contention that degrades overall throughput

Data Locality and Transfer Costs: Large-scale ETL systems must consider data gravity - the tendency for computation to move toward data rather than vice versa. Moving terabytes across network boundaries is expensive in both time and money.

Data Movement Pattern	Cost Factor	Optimization Strategy	Trade-off
Cross-Region Transfer	Network egress charges, latency	Regional data replication	Storage cost vs. transfer cost
Cross-Cloud Transfer	Bandwidth costs, vendor lock-in	Multi-cloud data mesh	Complexity vs. flexibility
On-Premise to Cloud	Internet bandwidth limits	Incremental sync, compression	Migration time vs. operational cost
Database to Processing	Query overhead, connection limits	Result caching, read replicas	Freshness vs. performance

> **Scalability Design Principle**: The system architecture must support **elastic scaling** that automatically adjusts resource allocation based on pipeline demand and data volume patterns. This requires sophisticated workload prediction and resource provisioning that considers both current demand and scheduled pipeline execution.

Implementation Guidance

Technology Recommendations

The architecture described above can be implemented using various technology stacks. The choice depends on organizational constraints, performance requirements, and operational preferences.

Component	Simple Option	Advanced Option	Enterprise Option
Pipeline Definition	YAML configuration files	Python-based DSL (Airflow-style)	Visual pipeline builder with code generation
Task Execution	Process-based execution	Container orchestration (Docker)	Kubernetes jobs with resource isolation
State Management	SQLite with WAL mode	PostgreSQL with connection pooling	Distributed database (CockroachDB, Spanner)
Message Queue	In-memory queues	Redis with persistence	Apache Kafka or AWS SQS
Monitoring	File-based logging	Structured logging (ELK stack)	APM integration (Datadog, New Relic)
Failure Recovery	Simple retry with backoff	Dead letter queues	Circuit breakers with health checks

Core Architecture Patterns

Repository Pattern for Pipeline Definitions: Abstracts pipeline storage and versioning to support multiple configuration sources.

PYTHON

```python
from abc import ABC, abstractmethod
from typing import List, Optional
from dataclasses import dataclass
from datetime import datetime

@dataclass
class PipelineDefinition:
    """Core pipeline definition with metadata and task specifications."""
    id: str
    name: str
    description: str
    schedule: str  # cron expression
    tasks: List['TaskDefinition']
    parameters: dict
    created_at: datetime
    version: int

@dataclass
class TaskDefinition:
    """Individual task within a pipeline with dependencies and configuration."""
    id: str
    name: str
    type: str  # extract, transform, load
    config: dict
    dependencies: List[str]  # upstream task IDs
    retry_policy: 'RetryPolicy'
    timeout_seconds: int

@dataclass
class RetryPolicy:
    """Configurable retry behavior for task failures."""
    max_attempts: int
    backoff_seconds: int
    exponential_backoff: bool
```

```python
    retry_on_error_types: List[str]

class PipelineRepository(ABC):
    """Abstract interface for pipeline definition storage and retrieval."""

    @abstractmethod
    def get_pipeline(self, pipeline_id: str) -> Optional[PipelineDefinition]:
        """Retrieve pipeline definition by ID."""
        pass

    @abstractmethod
    def list_pipelines(self) -> List[PipelineDefinition]:
        """List all available pipeline definitions."""
        pass

    @abstractmethod
    def save_pipeline(self, pipeline: PipelineDefinition) -> None:
        """Persist pipeline definition with version increment."""
        pass

    @abstractmethod
    def validate_pipeline(self, pipeline: PipelineDefinition) -> List[str]:
        """Validate pipeline definition and return error messages."""
        pass

class YamlPipelineRepository(PipelineRepository):
    """File-based pipeline repository using YAML configuration files."""

    def __init__(self, config_directory: str):
        self.config_directory = config_directory
        # TODO: Initialize YAML parser and file system monitoring
        # TODO: Implement configuration file validation schema
        # TODO: Set up file watcher for automatic reload on changes
```

```
    def get_pipeline(self, pipeline_id: str) -> Optional[PipelineDefinition]:
        # TODO: Load YAML file for specified pipeline ID
        # TODO: Parse YAML content into PipelineDefinition object
        # TODO: Validate task dependency references within pipeline
        # TODO: Return None if file not found or parsing fails
        pass

    def validate_pipeline(self, pipeline: PipelineDefinition) -> List[str]:
        # TODO: Check for circular dependencies in task graph
        # TODO: Validate task configuration schemas by type
        # TODO: Ensure all dependency references point to valid tasks
        # TODO: Validate cron schedule expression syntax
        # TODO: Check for duplicate task IDs within pipeline
        pass
```

State Machine Pattern for Task Execution: Manages task lifecycle transitions with clear state boundaries and event handling.

PYTHON

```python
from enum import Enum

from typing import Dict, Callable, Optional

import logging


class TaskState(Enum):

    """Enumeration of all possible task execution states."""

    PENDING = "pending"

    WAITING = "waiting"      # waiting for dependencies

    QUEUED = "queued"        # ready to execute

    RUNNING = "running"

    SUCCESS = "success"

    FAILED = "failed"

    RETRYING = "retrying"

    CANCELLED = "cancelled"

    SKIPPED = "skipped"      # skipped due to upstream failure


class TaskEvent(Enum):

    """Events that trigger task state transitions."""

    DEPENDENCIES_MET = "dependencies_met"

    EXECUTION_STARTED = "execution_started"

    EXECUTION_COMPLETED = "execution_completed"

    EXECUTION_FAILED = "execution_failed"

    RETRY_SCHEDULED = "retry_scheduled"

    MAX_RETRIES_EXCEEDED = "max_retries_exceeded"

    CANCELLED_BY_USER = "cancelled_by_user"

    UPSTREAM_FAILED = "upstream_failed"


@dataclass

class TaskExecution:

    """Runtime state for individual task execution."""

    task_id: str

    pipeline_run_id: str

    state: TaskState

    attempt_count: int
```

```
    started_at: Optional[datetime]
    completed_at: Optional[datetime]
    error_message: Optional[str]
    logs: List[str]
    metrics: Dict[str, float]

class TaskStateMachine:
    """Manages task state transitions and enforces valid state changes."""

    # Valid state transitions - current_state -> {event -> next_state}
    TRANSITIONS: Dict[TaskState, Dict[TaskEvent, TaskState]] = {
        TaskState.PENDING: {
            TaskEvent.DEPENDENCIES_MET: TaskState.QUEUED,
            TaskEvent.UPSTREAM_FAILED: TaskState.SKIPPED,
            TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED
        },
        TaskState.QUEUED: {
            TaskEvent.EXECUTION_STARTED: TaskState.RUNNING,
            TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED
        },
        TaskState.RUNNING: {
            TaskEvent.EXECUTION_COMPLETED: TaskState.SUCCESS,
            TaskEvent.EXECUTION_FAILED: TaskState.FAILED,
            TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED
        },
        TaskState.FAILED: {
            TaskEvent.RETRY_SCHEDULED: TaskState.RETRYING,
            TaskEvent.MAX_RETRIES_EXCEEDED: TaskState.FAILED
        },
        TaskState.RETRYING: {
            TaskEvent.EXECUTION_STARTED: TaskState.RUNNING
        }
    }
```

```
    def __init__(self):
        self.logger = logging.getLogger(__name__)
        # TODO: Initialize state transition hooks for monitoring
        # TODO: Set up metrics collection for state change events
        # TODO: Configure audit logging for compliance requirements

    def transition(self, execution: TaskExecution, event: TaskEvent,
                   error_message: Optional[str] = None) -> bool:
        """
        Attempt to transition task to new state based on event.
        Returns True if transition was successful, False otherwise.
        """
        # TODO: Validate current state allows this event transition
        # TODO: Update execution object with new state and metadata
        # TODO: Log state transition with timestamp and context
        # TODO: Trigger any registered state change hooks
        # TODO: Update metrics counters for monitoring dashboard
        # TODO: Return False if transition is invalid for current state
        pass

    def get_valid_events(self, current_state: TaskState) -> List[TaskEvent]:
        """Return list of events that can be applied to current state."""
        # TODO: Look up valid transitions from TRANSITIONS mapping
        # TODO: Return empty list if state has no valid transitions
        pass
```

Plugin System for Extensible Connectors: Enables adding new data sources and destinations without modifying core system.

PYTHON

```python
class DataConnector(ABC):
    """Base class for all data source and destination connectors."""

    @abstractmethod
    def connect(self, config: dict) -> 'Connection':
        """Establish connection to data source using provided configuration."""
        pass

    @abstractmethod
    def extract(self, connection: 'Connection', query: dict) -> 'DataStream':
        """Extract data from source system based on query parameters."""
        pass

    @abstractmethod
    def load(self, connection: 'Connection', data: 'DataStream',
             target: str) -> 'LoadResult':
        """Load data stream to target destination."""
        pass

    @abstractmethod
    def validate_config(self, config: dict) -> List[str]:
        """Validate connector configuration and return error messages."""
        pass

class DatabaseConnector(DataConnector):
    """Generic database connector supporting SQL-based sources."""

    def __init__(self, driver_name: str):
        self.driver_name = driver_name
        # TODO: Initialize database driver and connection pooling
        # TODO: Set up query timeout and retry configuration
        # TODO: Configure connection health checking
```

```
    def extract(self, connection: 'Connection', query: dict) -> 'DataStream':
        # TODO: Build SQL query from query parameters
        # TODO: Execute query with cursor-based pagination for large results
        # TODO: Handle connection failures with automatic retry
        # TODO: Stream results to avoid memory exhaustion
        # TODO: Track extraction metrics (rows processed, query time)
        pass

    def load(self, connection: 'Connection', data: 'DataStream',
             target: str) -> 'LoadResult':
        # TODO: Begin database transaction for atomicity
        # TODO: Batch insert records for optimal performance
        # TODO: Handle constraint violations and data type errors
        # TODO: Commit transaction only after all batches succeed
        # TODO: Record load statistics and performance metrics
        pass

class ConnectorRegistry:
    """Registry for discovering and instantiating data connectors."""

    def __init__(self):
        self._connectors: Dict[str, type] = {}
        # TODO: Scan for connector implementations in plugin directories
        # TODO: Validate connector implementations match interface
        # TODO: Set up hot-reload capability for development environments

    def register_connector(self, name: str, connector_class: type) -> None:
        # TODO: Validate connector_class implements DataConnector interface
        # TODO: Store connector in registry with name as key
        # TODO: Log connector registration for debugging
        pass

    def get_connector(self, name: str, config: dict) -> DataConnector:
```

```
        # TODO: Look up connector class by name in registry
        # TODO: Instantiate connector with provided configuration
        # TODO: Validate configuration before returning instance
        # TODO: Raise descriptive error if connector not found
        pass
```

Development Environment Setup

Recommended Project Structure:

```
etl-pipeline-system/
├── src/
│   ├── core/
│   │   ├── __init__.py
│   │   ├── pipeline.py          # PipelineDefinition, TaskDefinition models
│   │   ├── repository.py        # PipelineRepository interface
│   │   ├── state_machine.py     # TaskStateMachine implementation
│   │   └── executor.py          # Task execution engine
│   ├── connectors/
│   │   ├── __init__.py
│   │   ├── base.py              # DataConnector abstract base class
│   │   ├── database.py          # DatabaseConnector implementation
│   │   ├── file_system.py       # File-based extraction/loading
│   │   └── rest_api.py          # HTTP REST API connector
│   ├── scheduling/
│   │   ├── __init__.py
│   │   ├── scheduler.py         # Cron-based pipeline scheduling
│   │   ├── dag_parser.py        # DAG validation and topological sorting
│   │   └── dependency_resolver.py  # Task dependency management
│   ├── monitoring/
│   │   ├── __init__.py
│   │   ├── metrics.py           # Performance metrics collection
│   │   ├── logging.py           # Structured logging configuration
│   │   └── alerting.py          # Failure notification system
│   └── web/
│       ├── __init__.py
│       ├── api.py               # REST API for pipeline management
│       ├── dashboard.py         # Web dashboard for monitoring
│       └── templates/           # HTML templates for UI
├── tests/
│   ├── unit/                    # Component-specific unit tests
│   ├── integration/             # End-to-end pipeline tests
│   └── fixtures/                # Test data and mock configurations
├── config/
│   ├── pipelines/               # Pipeline definition YAML files
│   ├── connectors.yaml          # Data source connection configs
│   └── scheduler.yaml           # Scheduling and retry policies
├── docs/
│   ├── api/                     # API documentation
│   ├── connectors/              # Connector development guides
│   └── deployment/              # Operational deployment guides
├── requirements.txt             # Python dependencies
├── setup.py                     # Package installation configuration
└── README.md                    # Quick start and architecture overview
```

Language-Specific Implementation Notes

Python Dependency Management: Use `requirements.txt` for development and `setup.py` for distribution. Pin exact versions for reproducible builds.

Concurrency Patterns: Leverage `asyncio` for I/O-bound operations (database queries, API calls) and `multiprocessing` for CPU-bound transformations. Use `concurrent.futures.ThreadPoolExecutor` for mixed workloads.

Configuration Management: Use `pydantic` for configuration validation with automatic type conversion and detailed error messages. Support both environment variables and configuration files.

Logging Best Practices: Configure structured logging with `python-json-logger` for machine-readable logs. Include correlation IDs to trace requests across distributed components.

Error Handling Patterns: Create custom exception hierarchies for different failure types (transient vs. permanent). Use `tenacity` library for declarative retry policies with exponential backoff.

Milestone Checkpoints

Context Validation Checkpoint: After reading this section, you should be able to:

1. **Explain the Assembly Line Analogy**: Describe how ETL pipelines resemble manufacturing processes and why dependency management is the central challenge.
2. **Compare ETL Platforms**: Create a comparison table showing when to choose Airflow vs. Dagster vs. custom development based on specific requirements.
3. **Identify Technical Challenges**: Given a data processing scenario, identify which of the four core challenges (dependency management, failure recovery, data consistency, scalability) apply and why.
4. **Design Trade-off Analysis**: For any architectural decision, articulate the options considered, decision made, rationale, and consequences using the ADR format.

Verification Steps:

- Draw a simple ETL pipeline as a DAG and identify potential failure points
- Write a brief architectural decision record for choosing between different state management approaches
- List the key differences between task-centric (Airflow) and data-centric (Dagster) pipeline models
- Explain why idempotency is crucial for ETL operations with specific examples

Common Issues at This Stage:

- **Symptom**: Overwhelmed by the complexity of existing ETL tools
- **Cause**: Trying to understand implementation details before architectural concepts
- **Fix**: Focus on the mental models and analogies first, then gradually add technical detail
- **Symptom**: Unclear about build vs. buy decision criteria
- **Cause**: Not considering total cost of ownership including maintenance and feature development
- **Fix**: Create a decision matrix weighing development cost, operational overhead, and strategic flexibility

Goals and Non-Goals

> **Milestone(s):** Foundation for all milestones - defines system boundaries and requirements that guide implementation decisions across pipeline definition, data processing, and orchestration features.

Mental Model: Project Charter

Think of this goals section as a **project charter for a construction project**. Before breaking ground on a skyscraper, architects and stakeholders must agree on fundamental questions: How many floors? What's the budget? Will it have a parking garage? What about a helicopter pad? The charter explicitly states "we're building a 40-story office building with underground parking" and equally importantly

"we're NOT building a shopping mall or residential units." This prevents scope creep and ensures everyone builds toward the same vision.

Similarly, our ETL system goals define both what we're building and what we're deliberately not building. Without clear boundaries, an ETL system could expand infinitely - real-time streaming, machine learning pipelines, data lakes, visualization tools, user management systems. The goals act as guardrails that keep the architecture focused and implementable within reasonable complexity bounds.

Functional Goals

The functional goals define the core capabilities our ETL system must deliver to users. These represent the essential features that justify the system's existence and differentiate it from simple scripts or manual processes.

Pipeline Definition and Management

Our system must enable users to define complex data workflows as directed acyclic graphs (DAGs) where tasks have dependencies on other tasks. Users should be able to express these dependencies declaratively using configuration files rather than writing procedural code. The system must support both YAML and Python configuration formats to accommodate different user preferences and complexity levels.

The pipeline definition capability includes parameter substitution, allowing users to create reusable pipeline templates that can be instantiated with different configuration values. For example, a data ingestion pipeline template might accept database connection parameters and table names as runtime variables, enabling the same pipeline logic to process different data sources.

> **Decision: Declarative Pipeline Definition**
>
> - **Context**: Users need to define complex data workflows with task dependencies. Options include procedural code, visual designers, or declarative configuration.
> - **Options Considered**: Pure Python code, GUI-based visual editor, YAML/JSON configuration files
> - **Decision**: Declarative YAML/Python configuration with programmatic task definition support
> - **Rationale**: Declarative formats enable version control, code review, and automated validation. Visual editors create vendor lock-in and are difficult to version control. Pure code provides flexibility but requires programming expertise.
> - **Consequences**: Enables infrastructure-as-code practices and makes pipelines reviewable by non-programmers, but requires building a custom DSL and validation layer.

Feature	Description	User Benefit
DAG Definition	Define task dependencies as directed acyclic graphs	Ensures proper execution order and enables parallel processing
Parameter Substitution	Runtime variable replacement in configurations	Creates reusable pipeline templates for different environments
Multi-format Support	Both YAML and Python configuration options	Accommodates different user skill levels and complexity needs
Dependency Validation	Automatic cycle detection and dependency verification	Prevents invalid pipeline definitions before execution

Data Extraction and Loading

The system must provide robust connectors for extracting data from common enterprise data sources including relational databases, REST APIs, and file systems. These connectors must handle authentication, connection pooling, and error recovery automatically while exposing simple configuration interfaces to users.

For databases, the extraction capability must support both full table dumps and incremental loading strategies. Incremental loading uses techniques like watermarking (tracking the maximum timestamp or ID processed) and change data capture to extract only new or

modified records since the last pipeline run. This dramatically reduces processing time and resource usage for large datasets.

Loading capabilities must support bulk insertion strategies that optimize throughput for target systems. The system should automatically batch records, use appropriate bulk loading APIs (like PostgreSQL's COPY command), and handle schema mapping between source and destination systems when column names or types differ.

Connector Type	Extraction Features	Loading Features	Incremental Support
Database	SQL query execution, connection pooling, authentication	Bulk inserts, upserts, schema mapping	Watermark-based, CDC integration
REST API	Pagination, rate limiting, retry logic, authentication	HTTP POST/PUT with batching	Cursor-based pagination, timestamp filtering
File System	Multiple formats (CSV, JSON, Parquet), compression	Atomic writes, partitioning, compression	File modification time, filename patterns

Data Transformation and Validation

The transformation engine must support both SQL-based transformations for users familiar with declarative data manipulation and Python user-defined functions (UDFs) for custom business logic that cannot be expressed in SQL. SQL transformations should support templating to inject runtime parameters and enable reusable transformation logic.

Data validation capabilities must enforce schema constraints, business rules, and data quality checks during the transformation process. The system should support configurable validation strategies - some pipelines may reject invalid records entirely, while others may flag invalid records for manual review but continue processing valid data.

Schema evolution handling is critical for production systems where source data structures change over time. The transformation engine must detect schema changes, apply configurable evolution strategies (like adding default values for new columns), and maintain backward compatibility with existing pipeline definitions.

> **Decision: Hybrid SQL and Python Transformation Support**
>
> - **Context**: Users have varying technical backgrounds and transformation complexity requirements. Some transformations are simple aggregations while others require complex business logic.
> - **Options Considered**: SQL-only, Python-only, or hybrid approach supporting both
> - **Decision**: Support both SQL and Python transformations with seamless integration
> - **Rationale**: SQL handles 80% of common transformations efficiently and is familiar to analysts. Python provides flexibility for complex logic and integration with external libraries.
> - **Consequences**: Increases implementation complexity but maximizes user adoption by supporting different skill sets and use cases.

Pipeline Orchestration and Scheduling

The orchestration engine must execute pipelines according to user-defined schedules using standard cron expressions for time-based triggers. Additionally, the system must support event-driven execution where pipelines trigger automatically when upstream data sources change or external events occur.

Task execution must handle parallelism intelligently, running independent tasks concurrently while respecting dependency constraints. The system should provide configurable resource limits to prevent any single pipeline from overwhelming the execution environment.

Failure handling capabilities must include configurable retry policies with exponential backoff, dead letter queues for persistently failing tasks, and alerting integration to notify operators when manual intervention is required. The system must maintain detailed execution history and provide log aggregation for debugging failed pipeline runs.

Non-Functional Goals

Non-functional goals define the quality attributes and operational characteristics the system must exhibit in production environments. These requirements significantly influence architectural decisions and implementation approaches.

Performance and Scalability

The system must process datasets ranging from thousands to millions of records efficiently. Pipeline execution overhead should remain minimal - a simple two-task pipeline should complete within 30 seconds of trigger time, with most overhead attributed to actual data processing rather than orchestration.

Task parallelism must scale to utilize available CPU cores effectively. On a 4-core development machine, four independent tasks should execute concurrently. The architecture must support horizontal scaling where additional worker machines can be added to increase overall processing capacity.

Memory usage must remain bounded even for large datasets. The system should use streaming and batching strategies to process datasets larger than available RAM. A pipeline processing a 1GB dataset should not require more than 256MB of heap memory at peak usage.

Performance Metric	Target	Measurement Method
Pipeline Startup Overhead	< 30 seconds for simple pipelines	Time from trigger to first task execution
Task Parallelism	Utilize all available CPU cores	Concurrent task execution count
Memory Efficiency	Process 4x data size in available RAM	Peak memory usage vs dataset size
Throughput	10,000 records/second for simple transformations	Records processed per unit time

Reliability and Availability

The system must handle infrastructure failures gracefully without losing data or leaving pipelines in inconsistent states. Task execution must be idempotent - running the same pipeline multiple times should produce identical results without negative side effects. This enables safe retries and recovery from partial failures.

State persistence must survive process restarts and machine failures. Pipeline execution state, task progress, and metadata must be stored in durable storage with appropriate backup and recovery procedures. The system should resume interrupted pipelines from the last successful checkpoint rather than restarting from the beginning.

Error recovery must be automatic for transient failures like network timeouts or temporary resource unavailability. Only persistent errors that require human intervention should halt pipeline execution and trigger alerts.

> **Decision: Checkpoint-Based Recovery Strategy**
>
> - **Context**: Pipelines may fail partway through execution due to infrastructure issues, and restarting from the beginning wastes computational resources.
> - **Options Considered**: No recovery (restart from beginning), task-level checkpointing, operation-level checkpointing
> - **Decision**: Task-level checkpointing with idempotent operation design
> - **Rationale**: Task-level granularity provides good balance between implementation complexity and recovery efficiency. Operation-level checkpointing adds significant complexity for marginal benefit.
> - **Consequences**: Failed pipelines can resume from the last completed task, but individual tasks must be designed to be idempotent and handle partial state.

Monitoring and Observability

The system must provide comprehensive visibility into pipeline execution through metrics, logs, and tracing. Users should be able to answer questions like "Why did my pipeline fail?" and "Which task is the bottleneck?" without accessing raw log files or debugging

tools.

Real-time monitoring must track pipeline run status, task execution progress, and system resource utilization. Historical metrics should enable trend analysis to identify performance degradation or capacity planning needs. The monitoring system must integrate with standard observability tools like Prometheus, Grafana, or cloud-native monitoring services.

Data lineage tracking must capture the provenance of every dataset - which source systems contributed data, what transformations were applied, and when the processing occurred. This lineage information supports compliance requirements, debugging, and impact analysis when upstream data sources change.

Observability Feature	Information Provided	Use Case
Real-time Status	Current pipeline and task execution state	Operations monitoring and alerting
Performance Metrics	Execution time, throughput, resource usage	Performance optimization and capacity planning
Data Lineage	Source-to-destination data flow tracking	Compliance, debugging, impact analysis
Audit Logs	User actions and system changes	Security auditing and change tracking

Explicit Non-Goals

Explicitly defining what the system will not do is equally important as defining what it will do. These non-goals prevent scope creep and help users understand the system's boundaries and integration requirements.

Real-Time Stream Processing

This ETL system focuses on batch processing workflows and does not attempt to provide real-time stream processing capabilities. Users requiring sub-second data processing latency should integrate with dedicated streaming platforms like Apache Kafka, Apache Flink, or cloud streaming services.

The system's scheduling granularity targets minute-level intervals rather than millisecond-level responsiveness. While the system supports event-driven pipeline triggers, these events are expected to be coarse-grained notifications (like "new file arrived" or "database sync completed") rather than individual record-level events.

This architectural decision significantly simplifies the system design by avoiding the complexity of stream processing semantics, windowing, watermarking, and exactly-once processing guarantees that streaming systems require.

Data Storage and Serving

The ETL system acts as a data processing orchestrator rather than a data storage platform. Users must provide their own source and destination systems - the ETL system does not include embedded databases, data lakes, or serving layers.

This boundary means the system does not handle data modeling, query optimization, or end-user data access patterns. Users requiring these capabilities must integrate with appropriate storage solutions like data warehouses, operational databases, or analytical platforms.

The system focuses on the "T" (Transform) and orchestration aspects of ETL while delegating the "E" (Extract) and "L" (Load) to specialized connectors that interface with existing storage systems.

Visual Pipeline Designer

While the system provides pipeline visualization for monitoring and debugging purposes, it does not include a drag-and-drop visual pipeline designer or GUI-based pipeline authoring tools. Pipeline definitions must be created using configuration files or programmatic APIs.

This decision prioritizes version control, code review, and infrastructure-as-code practices over visual ease-of-use. Visual pipeline designers often create vendor lock-in and make it difficult to apply software engineering best practices to pipeline definitions.

Users requiring visual authoring capabilities should use external tools that can generate compatible pipeline configurations or integrate with the system's programmatic APIs.

Machine Learning and Advanced Analytics

The system provides general-purpose data transformation capabilities but does not include specialized machine learning features like model training, inference, or ML-specific data preprocessing operations. Users requiring these capabilities should integrate with dedicated ML platforms or data science tools.

This boundary prevents the system from becoming an overly complex platform that tries to solve every data-related problem. By focusing on general ETL orchestration, the system can integrate effectively with specialized tools rather than competing with them.

> **Decision: Integration-First Architecture Over Platform Expansion**
>
> - **Context**: Data ecosystems require many specialized tools (streaming, ML, storage, visualization). The system could try to include these capabilities or focus on integration.
> - **Options Considered**: Build an all-in-one data platform, focus purely on ETL orchestration, hybrid approach with some built-in capabilities
> - **Decision**: Focus on ETL orchestration with excellent integration capabilities
> - **Rationale**: All-in-one platforms become bloated and cannot compete with specialized tools. Integration-first design enables users to choose best-of-breed tools for each use case.
> - **Consequences**: Users must manage multiple tools but can optimize each component. System remains focused and maintainable.

Multi-Tenancy and User Management

The system does not provide built-in user authentication, authorization, or multi-tenant isolation capabilities. Deployment security and access control must be handled by the surrounding infrastructure using standard practices like network isolation, service authentication, and infrastructure-level access controls.

This decision reduces system complexity and allows organizations to integrate with their existing identity and access management solutions rather than learning yet another authentication system. The system assumes it operates in a trusted environment where access control is enforced at the infrastructure layer.

High-Frequency Trading or Financial Data

While the system can process financial datasets, it does not provide specialized features for high-frequency trading, market data processing, or real-time risk management that require microsecond-level latency and specialized financial protocols.

The system's reliability and consistency guarantees target business intelligence and analytical use cases where eventual consistency and minute-level latency are acceptable, rather than trading systems where milliseconds matter and regulatory compliance requires specialized audit trails.

Non-Goal Category	What We Don't Build	Recommended Alternatives
Stream Processing	Real-time event processing, windowing, exactly-once semantics	Apache Kafka, Flink, cloud streaming services
Data Storage	Embedded databases, data lakes, query engines	PostgreSQL, Snowflake, BigQuery, S3/HDFS
Visual Authoring	Drag-and-drop pipeline designer, GUI configuration	External tools that generate configurations
Machine Learning	Model training, inference, ML-specific preprocessing	MLflow, Kubeflow, cloud ML platforms
User Management	Authentication, authorization, multi-tenancy	Infrastructure-level access controls, identity providers
Financial Systems	High-frequency trading, microsecond latency, specialized compliance	Dedicated financial data platforms

Implementation Guidance

This implementation guidance helps translate the functional and non-functional goals into concrete technical decisions and development practices.

Technology Recommendations

The following technology choices support the goals defined above while balancing simplicity for development with production-readiness for deployment:

Component	Simple Option	Advanced Option	Goal Alignment
Configuration Format	YAML with JSON Schema validation	Python-based DSL with type checking	Supports declarative pipeline definition goal
Task Execution	Thread pool with concurrent.futures	Distributed task queue (Celery, RQ)	Enables parallelism and scalability goals
State Storage	SQLite for development	PostgreSQL for production	Provides reliability and state persistence
Logging	Python logging with file handlers	Structured logging with centralized collection	Supports observability and monitoring goals
Scheduling	APScheduler for cron support	Integration with external schedulers (Airflow, etc.)	Enables time-based and event-driven execution
Monitoring	Simple metrics collection	Prometheus + Grafana integration	Provides production-ready observability

Recommended Project Structure

Organize the codebase to support the modular architecture implied by our functional goals:

```
etl-pipeline/
├── src/
│   ├── pipeline/
│   │   ├── __init__.py
│   │   ├── definition.py          # PipelineDefinition, TaskDefinition classes
│   │   ├── validation.py          # Pipeline validation logic
│   │   └── serialization.py       # YAML/JSON config parsing
│   ├── execution/
│   │   ├── __init__.py
│   │   ├── scheduler.py           # Schedule management and triggering
│   │   ├── executor.py            # Task execution engine
│   │   └── state_machine.py       # TaskState transitions
│   ├── connectors/
│   │   ├── __init__.py
│   │   ├── base.py               # Abstract connector interfaces
│   │   ├── database.py           # Database extraction/loading
│   │   ├── api.py                # REST API connectors
│   │   └── filesystem.py         # File-based connectors
│   ├── transforms/
│   │   ├── __init__.py
│   │   ├── sql_transform.py      # SQL-based transformations
│   │   └── python_transform.py   # Python UDF support
│   └── monitoring/
│       ├── __init__.py
│       ├── metrics.py            # Performance and execution metrics
│       ├── logging.py            # Log aggregation and formatting
│       └── lineage.py            # Data lineage tracking
├── config/
│   ├── pipeline_schemas/         # JSON schemas for pipeline validation
│   └── examples/                 # Example pipeline definitions
├── tests/
│   ├── unit/                    # Component-level unit tests
│   ├── integration/             # End-to-end pipeline tests
│   └── fixtures/                # Test data and mock configurations
└── docs/
    ├── user_guide.md            # Pipeline authoring documentation
    └── api_reference.md         # Programmatic API documentation
```

Core Data Structures Starter Code

PYTHON

```python
from dataclasses import dataclass
from datetime import datetime
from typing import List, Dict, Optional, Any
from enum import Enum

@dataclass
class RetryPolicy:
    """Defines retry behavior for failed task executions."""
    max_attempts: int
    backoff_seconds: int
    exponential_backoff: bool
    retry_on_error_types: List[str]

@dataclass
class TaskDefinition:
    """Defines a single task within a pipeline DAG."""
    id: str
    name: str
    type: str  # 'extract', 'transform', 'load'
    config: Dict[str, Any]
    dependencies: List[str]  # Task IDs this task depends on
    retry_policy: RetryPolicy
    timeout_seconds: int

@dataclass
class PipelineDefinition:
    """Complete pipeline specification with metadata and task definitions."""
    id: str
    name: str
    description: str
    schedule: str  # Cron expression or event trigger
    tasks: List[TaskDefinition]
    parameters: Dict[str, Any]  # Runtime parameters
    created_at: datetime
```

```
    version: int


class TaskState(Enum):
    """Enumeration of possible task execution states."""
    PENDING = "pending"
    WAITING = "waiting"
    QUEUED = "queued"
    RUNNING = "running"
    SUCCESS = "success"
    FAILED = "failed"
    RETRYING = "retrying"
    CANCELLED = "cancelled"
    SKIPPED = "skipped"


class TaskEvent(Enum):
    """Events that trigger task state transitions."""
    DEPENDENCIES_MET = "dependencies_met"
    EXECUTION_STARTED = "execution_started"
    EXECUTION_COMPLETED = "execution_completed"
    EXECUTION_FAILED = "execution_failed"
    RETRY_SCHEDULED = "retry_scheduled"
    MAX_RETRIES_EXCEEDED = "max_retries_exceeded"
    CANCELLED_BY_USER = "cancelled_by_user"
    UPSTREAM_FAILED = "upstream_failed"


@dataclass
class TaskExecution:
    """Runtime state for a specific task execution instance."""
    task_id: str
    pipeline_run_id: str
    state: TaskState
    attempt_count: int
    started_at: Optional[datetime]
    completed_at: Optional[datetime]
```

```
    error_message: Optional[str]

    logs: List[str]

    metrics: Dict[str, float]
```

Configuration Validation Infrastructure

PYTHON

```python
import yaml

import jsonschema

from pathlib import Path

from typing import Optional, List


def get_pipeline(pipeline_id: str) -> Optional[PipelineDefinition]:

    """Retrieve pipeline definition by ID from configuration storage.



    Args:

        pipeline_id: Unique identifier for the pipeline



    Returns:

        PipelineDefinition if found, None otherwise



    TODO: Implement configuration storage backend (file, database, etc.)

    TODO: Add caching layer for frequently accessed pipelines

    TODO: Handle configuration versioning and schema migration

    """

    # Implementation will load from config storage

    pass


def validate_pipeline(pipeline: PipelineDefinition) -> List[str]:

    """Validate pipeline definition and return list of error messages.



    Args:

        pipeline: Pipeline definition to validate



    Returns:

        List of validation error messages (empty if valid)



    This function performs multiple validation checks:

    - Schema validation against JSON schema

    - DAG cycle detection

    - Task dependency reference validation
```

```
    - Resource and timeout constraint validation
    """
    errors = []

    # TODO 1: Validate against JSON schema for structural correctness
    # TODO 2: Check for cycles in task dependency graph
    # TODO 3: Verify all task dependencies reference existing tasks
    # TODO 4: Validate cron schedule expression syntax
    # TODO 5: Check resource constraints and timeout values
    # TODO 6: Validate connector configurations for each task type

    return errors

def transition(execution: TaskExecution, event: TaskEvent, error_message: Optional[str] = None) -> bool:
    """Attempt state transition based on event and current state.

    Args:
        execution: Current task execution instance
        event: Event triggering the transition
        error_message: Optional error details for failure events

    Returns:
        True if transition was valid and applied, False otherwise

    TODO: Implement state machine transition logic based on TRANSITIONS mapping
    TODO: Update execution timestamps and attempt counts
    TODO: Handle retry logic and backoff scheduling
    TODO: Log state changes for audit and debugging
    """
    # Implementation will check TRANSITIONS table and update execution state
    pass

# State transition mapping - defines valid state changes
TRANSITIONS = {
```

```
    TaskState.PENDING: {
        TaskEvent.DEPENDENCIES_MET: TaskState.QUEUED,
        TaskEvent.UPSTREAM_FAILED: TaskState.SKIPPED,
        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED,
    },
    TaskState.QUEUED: {
        TaskEvent.EXECUTION_STARTED: TaskState.RUNNING,
        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED,
    },
    TaskState.RUNNING: {
        TaskEvent.EXECUTION_COMPLETED: TaskState.SUCCESS,
        TaskEvent.EXECUTION_FAILED: TaskState.FAILED,
        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED,
    },
    TaskState.FAILED: {
        TaskEvent.RETRY_SCHEDULED: TaskState.RETRYING,
        TaskEvent.MAX_RETRIES_EXCEEDED: TaskState.FAILED,
    },
    TaskState.RETRYING: {
        TaskEvent.EXECUTION_STARTED: TaskState.RUNNING,
        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED,
    },
    # Success, cancelled, and skipped are terminal states
}
```

Milestone Checkpoint Guidelines

After implementing the goals and boundaries defined in this section, verify the following checkpoint behaviors:

1. **Goal Validation Checkpoint**: Create a simple pipeline definition and verify it can be loaded and validated without errors. The validation should catch basic issues like missing required fields or invalid cron expressions.
2. **Non-Goal Boundary Checkpoint**: Attempt to use features that are explicitly non-goals (like real-time streaming or visual authoring) and verify they are clearly unsupported with helpful error messages directing users to appropriate alternatives.
3. **Configuration Structure Checkpoint**: The project structure should accommodate the modular architecture implied by the functional goals. Each major component (pipeline definition, execution, connectors, transforms, monitoring) should have its own module with clear interfaces.

Common Implementation Pitfalls

⚠️ **Pitfall: Over-Engineering Goals** Early implementations often try to build every possible feature rather than focusing on the core goals. This leads to complex, unmaintainable systems that don't excel at their primary purpose. Stick to the defined functional goals and resist feature creep.

⚠️ **Pitfall: Ignoring Non-Functional Requirements** Functional goals are visible to users, but non-functional goals (performance, reliability, observability) often determine production success. Design data structures and interfaces with performance and monitoring in mind from the beginning rather than retrofitting these concerns later.

⚠️ **Pitfall: Unclear Goal Boundaries** Vague goal definitions lead to scope creep and architectural confusion. Each goal should be measurable and testable. If you can't write a test that verifies a goal is met, the goal needs to be more specific.

⚠️ **Pitfall: Non-Goals Become Goals** Teams often rationalize why a non-goal is actually essential and should be included. This dilutes the system's focus and increases complexity. When users request non-goal features, provide clear integration paths with external systems rather than expanding the system's scope.

High-Level Architecture

> **Milestone(s):** Foundation for all milestones - provides the architectural blueprint that guides implementation across pipeline definition (Milestone 1), data processing (Milestones 2-3), and orchestration/monitoring (Milestone 4).

Mental Model: Orchestra Conductor System

Think of our ETL system like a symphony orchestra with multiple specialized sections working in harmony. The **Conductor** (Scheduler) reads the musical score (pipeline definition) and coordinates when each section plays. The **Section Leaders** (Task Executors) manage their musicians and ensure they play their parts correctly. The **Stage Manager** (DAG Engine) ensures all the sheet music is correct and tells the conductor the proper sequence. The **Audio Engineers** (Monitoring System) record the performance, adjust levels, and alert everyone if something goes wrong. Finally, the **Musicians** themselves (Connectors) are the specialists who actually produce the music by extracting, transforming, and loading data.

Just as a conductor can't start the string section before the woodwinds finish their passage, our system ensures data dependencies are respected. When a musician makes a mistake, the section leader (executor) can have them retry their part without stopping the entire orchestra. The audio engineers continuously monitor the performance quality and can quickly identify which section needs attention.

This orchestration analogy helps us understand why we need separate, specialized components rather than a monolithic system - each component has a distinct responsibility, and the coordination between them creates the beautiful symphony of data processing.

System Components

Our ETL system consists of five core components, each with distinct responsibilities and clear boundaries. Understanding these components and their relationships is crucial for implementing a maintainable and scalable data pipeline system.

DAG Definition Engine

The **DAG Definition Engine** serves as the foundation of our pipeline system, responsible for parsing, validating, and maintaining pipeline definitions. This component acts as the "compiler" for our ETL pipelines, taking human-readable pipeline configurations and transforming them into executable task graphs.

Component Responsibility	Description	Key Data Structures
Pipeline Parsing	Reads YAML/Python configuration files and converts them into `PipelineDefinition` objects	`PipelineDefinition`, `TaskDefinition`
Dependency Validation	Performs cycle detection and validates task dependency relationships	DAG validation results, dependency graphs
Execution Graph Building	Creates optimized task execution graphs using topological sorting	Execution order lists, dependency matrices
Schema Validation	Ensures pipeline definitions conform to required schemas and constraints	Validation error collections

The DAG Engine maintains a registry of all pipeline definitions and provides versioning capabilities. When a pipeline definition changes, it validates the new version against existing data schemas and dependency constraints before allowing the update.

> **Design Insight**: The DAG Engine is intentionally stateless regarding pipeline executions. It only deals with pipeline *definitions*, not runtime state. This separation allows multiple scheduler instances to share the same pipeline definitions while maintaining independent execution state.

Pipeline Scheduler

The **Pipeline Scheduler** orchestrates when pipelines execute based on time-based schedules, external triggers, or data availability conditions. Think of it as the "air traffic control" system that manages the flow of pipeline executions across the entire system.

Scheduling Capability	Implementation Approach	Configuration Options
Cron-based Scheduling	Uses cron expressions parsed into trigger events	Standard cron syntax with seconds precision
Event-driven Triggers	Listens for external events (file arrivals, API calls)	HTTP webhooks, message queue listeners
Data Dependency Scheduling	Monitors upstream data sources for freshness	Watermark-based triggers, file modification monitoring
Manual Triggering	Provides API endpoints for on-demand execution	REST API with parameter overrides

The Scheduler maintains a priority queue of pending pipeline runs and coordinates with the Task Executor to ensure proper resource allocation. It implements backpressure mechanisms to prevent system overload when multiple pipelines are ready for execution.

Task Executor

The **Task Executor** is responsible for the actual execution of individual tasks within pipeline runs. It manages task lifecycle, resource allocation, failure handling, and state transitions. This component bridges the gap between high-level pipeline orchestration and low-level data processing operations.

Execution Capability	Implementation Details	Failure Handling
Parallel Task Execution	Manages thread pools with configurable concurrency limits	Task isolation prevents failures from affecting siblings
State Management	Tracks `TaskExecution` objects through `TaskState` transitions	Atomic state updates with event-driven notifications
Resource Allocation	Coordinates memory, CPU, and I/O resources across concurrent tasks	Resource limits with graceful degradation
Retry Logic	Implements exponential backoff based on `RetryPolicy` configurations	Configurable retry conditions and maximum attempt limits

The Task Executor maintains execution context for each task, including runtime parameters, connection pools, and intermediate results. It provides isolation between tasks to prevent resource conflicts and implements comprehensive logging for debugging purposes.

Data Connectors

Data Connectors are specialized components responsible for interfacing with external data systems. They provide a pluggable architecture that abstracts the complexity of different data sources and destinations behind a unified interface.

Connector Type	Supported Systems	Key Capabilities
Source Connectors	Databases (PostgreSQL, MySQL), APIs (REST, GraphQL), Files (S3, HDFS)	Incremental extraction, schema discovery, pagination handling
Destination Connectors	Data warehouses (Snowflake, BigQuery), Databases, File systems	Bulk loading, upsert operations, schema evolution
Transform Connectors	SQL engines (Spark, DuckDB), Python execution environments	Custom logic execution, data validation, aggregations

Each connector implements standardized interfaces for connection management, data streaming, and error handling. The connector architecture supports plugin-style extensibility, allowing new data sources to be added without modifying core system components.

Monitoring and Observability System

The **Monitoring and Observability System** provides comprehensive visibility into pipeline execution, performance metrics, and system health. It serves as the "control tower" that gives operators the information needed to maintain and optimize the ETL system.

Monitoring Aspect	Data Collected	Alerting Capabilities
Pipeline Execution	Run duration, success/failure rates, task timing	SLA violation alerts, failure notifications
Resource Utilization	Memory usage, CPU consumption, I/O throughput	Resource exhaustion warnings, capacity planning metrics
Data Quality	Record counts, schema validation results, data profiling	Data anomaly detection, quality threshold alerts
System Health	Component availability, error rates, dependency status	Service degradation alerts, dependency failure notifications

The monitoring system implements configurable alerting rules that can escalate issues based on severity and duration. It maintains historical metrics for trend analysis and capacity planning.

Component Interactions

Understanding how these components communicate and coordinate is essential for implementing a robust ETL system. Each interaction follows specific protocols and handles various failure scenarios gracefully.

Scheduler to DAG Engine Communication

The Scheduler queries the DAG Engine to retrieve pipeline definitions and validate execution requests. This interaction is read-only from the Scheduler's perspective, ensuring clear separation of concerns.

Interaction Type	Protocol	Error Handling
Pipeline Retrieval	Synchronous API call using `get_pipeline(pipeline_id)`	Returns `None` for non-existent pipelines, triggering schedule cleanup
Definition Validation	Calls `validate_pipeline(pipeline)` before scheduling runs	Validation errors prevent scheduling and generate operator alerts
Dependency Resolution	Requests task dependency graphs for execution planning	Invalid dependencies cause immediate run failure with detailed error messages

> **Architecture Decision: Pull-based Pipeline Retrieval**
>
> - **Context**: Scheduler needs access to current pipeline definitions for execution planning
> - **Options Considered**:
>   1. Push-based updates from DAG Engine to Scheduler
>   2. Pull-based retrieval with caching
>   3. Shared database access
> - **Decision**: Pull-based retrieval with local caching
> - **Rationale**: Provides better fault isolation, simpler failure scenarios, and allows independent scaling of components
> - **Consequences**: Slight latency for pipeline definition updates, but improved system reliability and reduced coupling

Scheduler to Task Executor Coordination

The Scheduler coordinates with Task Executors to manage pipeline run lifecycle and resource allocation. This relationship involves both command-style interactions and event-driven updates.

Coordination Activity	Message Format	Failure Recovery
Run Initialization	`PipelineRun` object with execution context and parameters	Executor startup failures trigger run cancellation and cleanup
Task Assignment	Individual `TaskExecution` assignments with dependency information	Task assignment failures cause graceful task skipping with downstream impact analysis
Progress Updates	Periodic state updates using `TaskState` transitions	Missing heartbeats trigger task timeout and potential retry scheduling
Resource Management	Resource allocation requests and availability notifications	Resource exhaustion causes task queuing with backpressure to Scheduler

The Scheduler maintains a registry of available Task Executor instances and implements load balancing across them. When an Executor becomes unavailable, the Scheduler redistributes pending tasks to healthy instances.

Task Executor to Connector Interactions

Task Executors invoke Data Connectors to perform actual data processing operations. These interactions involve streaming data transfers and require careful resource management.

Operation Type	Interface Method	Resource Considerations
Data Extraction	`extract(connection, query) -> DataStream`	Memory usage for result buffering, connection pool management
Data Loading	`load(connection, data, target) -> LoadResult`	Batch size optimization, transaction management
Schema Operations	`discover_schema(connection) -> SchemaInfo`	Connection timeout handling, metadata caching
Connection Management	`create_connection(config) -> Connection`	Connection pooling, credential management, network retry logic

Connectors implement streaming interfaces to handle large datasets without exhausting system memory. They provide detailed error information that the Task Executor can use for intelligent retry decisions.

Monitoring System Event Collection

The Monitoring System collects events and metrics from all other components through a combination of push-based events and pull-based metric collection.

Event Source	Event Types	Collection Method
Scheduler	Pipeline started, completed, failed, cancelled	Event streaming to monitoring message queue
Task Executor	Task state transitions, resource usage, performance metrics	Periodic metric export via HTTP endpoints
DAG Engine	Pipeline definition changes, validation errors	Audit log streaming with structured event format
Connectors	Data transfer volumes, connection health, schema changes	Embedded metric collection within execution context

The monitoring system implements event correlation to connect related activities across components. For example, it can trace a pipeline failure back through task failures to specific connector errors.

> **Design Insight**: Event correlation requires each pipeline run to carry a unique correlation ID that flows through all component interactions. This enables distributed tracing and comprehensive failure analysis.

Error Propagation and Recovery Coordination

When failures occur, components coordinate recovery efforts while maintaining system consistency and preventing cascading failures.

Failure Scenario	Detection Method	Recovery Coordination
Task Execution Failure	Executor reports failure via state transition	Scheduler evaluates retry policy and coordinates with fresh Executor instance
Connector Communication Loss	Connection timeout or error response	Executor implements circuit breaker pattern and reports degraded capability to Scheduler
Scheduler Instance Failure	Monitoring system detects missing heartbeats	Standby Scheduler instances take over using persistent state from shared storage
DAG Engine Unavailability	Pipeline retrieval failures	Scheduler operates with cached definitions and queues definition refresh requests

The system implements the **Circuit Breaker Pattern** for external dependencies. When a data source becomes unreliable, connectors temporarily stop attempting connections and return predictable errors rather than causing timeout delays across the system.

Deployment Topology

Understanding the recommended deployment architecture helps teams plan their infrastructure and avoid common deployment pitfalls. Our system supports both single-node development deployments and distributed production architectures.

Development Environment Structure

For development and testing, all components can run within a single process or as separate processes on a single machine. This configuration simplifies debugging while maintaining the same interfaces used in production.

```
project-root/
├── cmd/
│   ├── scheduler/          ← Scheduler service entry point
│   ├── executor/           ← Task Executor service entry point
│   └── single-node/        ← All-in-one development deployment
├── internal/
│   ├── dag/                ← DAG Definition Engine
│   │   ├── parser.go
│   │   ├── validator.go
│   │   └── graph.go
│   ├── scheduler/          ← Pipeline Scheduler
│   │   ├── cron_scheduler.go
│   │   ├── event_scheduler.go
│   │   └── run_manager.go
│   ├── executor/           ← Task Executor
│   │   ├── task_runner.go
│   │   ├── state_machine.go
│   │   └── resource_manager.go
│   ├── connectors/         ← Data Connectors
│   │   ├── database/
│   │   ├── api/
│   │   └── filesystem/
│   └── monitoring/         ← Monitoring System
│       ├── metrics.go
│       ├── events.go
│       └── alerting.go
├── configs/
│   ├── development.yaml    ← Single-node configuration
│   ├── staging.yaml        ← Multi-node staging environment
│   └── production.yaml     ← Production deployment configuration
└── pipelines/
    ├── examples/           ← Sample pipeline definitions
    └── schemas/            ← Pipeline definition JSON schemas
```

The development configuration runs all components in a single process with shared in-memory state. This enables rapid iteration and simplified debugging with standard development tools.

Production Deployment Architecture

Production deployments distribute components across multiple nodes for scalability, fault tolerance, and resource isolation. Each component type can scale independently based on workload characteristics.

Component Type	Scaling Strategy	Resource Requirements	High Availability
Scheduler	Active/Standby with shared persistent state	Low CPU, moderate memory for pipeline metadata	Shared state store (PostgreSQL/etcd) enables quick failover
Task Executor	Horizontal scaling with auto-scaling based on queue depth	High CPU/memory, varies by workload type	Stateless design allows dynamic scaling without data loss
DAG Engine	Load balanced read replicas with single writer	Low CPU, memory proportional to pipeline count	Git-based pipeline storage provides natural backup and versioning
Connectors	Embedded within Task Executors	Resource requirements vary by connector type	Connection pooling and retry logic handle transient failures
Monitoring	Clustered deployment with data replication	Moderate CPU, high storage for metrics retention	Time-series database clustering provides data durability

Network Communication Patterns

Components communicate using well-defined protocols that support both local and distributed deployments. The communication patterns prioritize reliability and observability over raw performance.

Communication Path	Protocol	Failure Handling	Security Considerations
Scheduler ↔ DAG Engine	HTTP REST with JSON	Retry with exponential backoff, circuit breaker	TLS encryption, API key authentication
Scheduler → Task Executor	Message queue (Redis/RabbitMQ)	Message acknowledgment, dead letter queue	Message-level encryption, queue access controls
Task Executor → Connectors	In-process function calls or HTTP	Exception handling, connection pooling	Credential injection, secret management
All Components → Monitoring	Event streaming + HTTP metrics	Best-effort delivery with local buffering	Metric anonymization, access logging

Architecture Decision: Message Queue for Task Assignment

- **Context**: Scheduler needs reliable way to assign tasks to available executors
- **Options Considered**:
  1. Direct HTTP calls to executor instances
  2. Shared database with polling
  3. Message queue with work distribution
- **Decision**: Message queue with work distribution pattern
- **Rationale**: Provides natural load balancing, handles executor failures gracefully, enables backpressure management
- **Consequences**: Introduces message queue as additional infrastructure dependency, but significantly improves system resilience

Persistent State Management

Each component manages different types of persistent state with appropriate storage technologies and consistency requirements.

Component	State Type	Storage Technology	Consistency Requirements
Scheduler	Run schedules, execution history	PostgreSQL with ACID transactions	Strong consistency for schedule integrity
Task Executor	Execution checkpoints, task logs	Object storage (S3) + local caching	Eventually consistent, focus on durability
DAG Engine	Pipeline definitions, validation cache	Git repository + local file cache	Strong consistency for definitions, eventual for cache
Monitoring	Metrics, events, alert history	Time-series database (Prometheus/InfluxDB)	Eventually consistent, optimized for write throughput

The system implements **checkpoint-based recovery** for long-running tasks. Task Executors periodically save execution state to persistent storage, enabling recovery from partial failures without reprocessing all data.

Security and Access Control Deployment

Production deployments implement defense-in-depth security with multiple layers of access control and data protection.

Security Layer	Implementation	Components Affected
Network Security	VPC isolation, security groups, TLS encryption	All inter-component communication
Authentication	Service-to-service authentication using certificates or tokens	Scheduler, DAG Engine, Monitoring APIs
Authorization	Role-based access control for pipeline operations	Pipeline definition access, execution permissions
Secret Management	External secret store integration (HashiCorp Vault, AWS Secrets Manager)	Connector configurations, database credentials
Audit Logging	Comprehensive logging of all system actions with immutable audit trail	All components with centralized log aggregation

Connector credentials are injected at runtime from the secret management system and never stored in pipeline definitions or system configuration files.

Common Pitfalls

⚠️ Pitfall: Shared State Between Components

Many developers initially try to share state directly between components (like using shared variables or files for coordination). This creates tight coupling and makes the system fragile to failures.

Why it's wrong: Shared state creates hidden dependencies that break when components are deployed separately or when failures occur. It also makes testing and development much more difficult.

How to fix it: Use message passing and well-defined APIs between components. Each component should own its state completely, and coordination should happen through explicit communication protocols.

⚠️ Pitfall: Synchronous Communication Everywhere

Using synchronous HTTP calls for all inter-component communication seems simple but creates cascading failure scenarios and tight coupling between component lifecycles.

Why it's wrong: When one component becomes slow or unavailable, synchronous calls cause the entire call chain to block or fail. This eliminates the benefits of component isolation.

How to fix it: Use asynchronous message queues for work distribution and events. Reserve synchronous calls only for queries where immediate responses are required (like pipeline definition retrieval).

⚠️ Pitfall: Insufficient Error Context

Developers often propagate generic errors between components without preserving context about what operation failed and why.

Why it's wrong: When debugging production issues, generic errors make it impossible to determine root causes. Operators need to understand which pipeline, task, and data source caused problems.

How to fix it: Implement structured error types that preserve operation context, correlation IDs, and relevant metadata. Each component should add its own context when propagating errors.

⚠️ Pitfall: No Resource Boundaries

Running all components with unlimited resource access causes resource contention and unpredictable performance as the system scales.

Why it's wrong: Without resource boundaries, one pipeline can consume all system memory or CPU, causing other pipelines to fail or perform poorly.

How to fix it: Implement resource limits at the Task Executor level, use connection pooling for external resources, and monitor resource usage to detect anomalies early.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Inter-Component Communication	HTTP REST with `requests` library	Message queues (Redis/RabbitMQ) with `celery` or `rq`
State Storage	SQLite for development, PostgreSQL for production	Distributed databases (PostgreSQL + Redis cluster)
Configuration Management	YAML files with `pyyaml` library	Configuration service (Consul, etcd) with hot reloading
Monitoring	Built-in Python `logging` + simple metrics	Prometheus + Grafana with custom metrics
Process Management	Single Python process with threading	Container orchestration (Docker + Kubernetes)

Recommended File Structure

```
etl-system/
├── src/
│   ├── etl/
│   │   ├── __init__.py
│   │   ├── core/                    ← Shared data structures and interfaces
│   │   │   ├── __init__.py
│   │   │   ├── models.py            ← PipelineDefinition, TaskDefinition, TaskExecution
│   │   │   ├── interfaces.py        ← Abstract base classes for connectors
│   │   │   └── events.py            ← TaskEvent, TaskState enums
│   │   ├── dag/                     ← DAG Definition Engine (Milestone 1)
│   │   │   ├── __init__.py
│   │   │   ├── parser.py            ← YAML/Python pipeline parsing
│   │   │   ├── validator.py         ← Cycle detection and validation
│   │   │   └── graph.py             ← Topological sorting and execution graphs
│   │   ├── scheduler/               ← Pipeline Scheduler (Milestone 4)
│   │   │   ├── __init__.py
│   │   │   ├── cron_scheduler.py    ← Time-based scheduling
│   │   │   ├── event_scheduler.py   ← Event-driven triggers
│   │   │   └── run_manager.py       ← Pipeline run lifecycle
│   │   ├── executor/                ← Task Executor (Milestone 4)
│   │   │   ├── __init__.py
│   │   │   ├── task_runner.py       ← Task execution engine
│   │   │   ├── state_machine.py     ← TaskState transitions
│   │   │   └── resource_manager.py  ← Resource allocation and limits
│   │   ├── connectors/              ← Data Connectors (Milestones 2-3)
│   │   │   ├── __init__.py
│   │   │   ├── base.py              ← Abstract connector interfaces
│   │   │   ├── database/            ← Database source/destination connectors
│   │   │   ├── api/                 ← REST API connectors
│   │   │   ├── filesystem/          ← File-based connectors
│   │   │   └── transforms/          ← SQL and Python transformation engines
│   │   └── monitoring/              ← Monitoring System (Milestone 4)
│   │       ├── __init__.py
│   │       ├── metrics.py           ← Metrics collection and export
│   │       ├── events.py            ← Event collection and correlation
│   │       └── alerting.py          ← Alert generation and notification
│   ├── scripts/
│   │   ├── start_scheduler.py       ← Scheduler service entry point
│   │   ├── start_executor.py        ← Executor service entry point
│   │   └── single_node.py           ← Development single-process runner
│   └── tests/
│       ├── unit/                    ← Component unit tests
│       ├── integration/             ← Cross-component integration tests
│       └── e2e/                     ← End-to-end pipeline tests
├── configs/
│   ├── development.yaml             ← Development environment configuration
│   ├── staging.yaml                 ← Staging environment configuration
│   └── production.yaml              ← Production deployment configuration
├── pipelines/
│   ├── examples/                    ← Sample pipeline definitions
│   │   ├── simple_etl.yaml
│   │   └── complex_analytics.yaml
│   └── schemas/
│       └── pipeline_schema.json     ← JSON Schema for pipeline validation
├── docker/
│   ├── Dockerfile.scheduler         ← Container image for scheduler
│   ├── Dockerfile.executor          ← Container image for executor
│   └── docker-compose.yml           ← Local development setup
└── docs/
    ├── architecture.md              ← This design document
    ├── api/                         ← Component API documentation
    └── examples/                    ← Usage examples and tutorials
```

Infrastructure Starter Code

Core Data Models (complete implementation):

PYTHON

```python
# src/etl/core/models.py

from dataclasses import dataclass, field

from datetime import datetime

from enum import Enum

from typing import List, Dict, Optional, Any

import uuid


@dataclass

class RetryPolicy:

    """Configuration for task retry behavior."""

    max_attempts: int = 3

    backoff_seconds: int = 60

    exponential_backoff: bool = True

    retry_on_error_types: List[str] = field(default_factory=lambda: ["ConnectionError", "TimeoutError"])


class TaskState(Enum):

    """All possible states for a task execution."""

    PENDING = "pending"

    WAITING = "waiting"

    QUEUED = "queued"

    RUNNING = "running"

    SUCCESS = "success"

    FAILED = "failed"

    RETRYING = "retrying"

    CANCELLED = "cancelled"

    SKIPPED = "skipped"


class TaskEvent(Enum):

    """Events that trigger task state transitions."""

    DEPENDENCIES_MET = "dependencies_met"

    EXECUTION_STARTED = "execution_started"

    EXECUTION_COMPLETED = "execution_completed"

    EXECUTION_FAILED = "execution_failed"

    RETRY_SCHEDULED = "retry_scheduled"
```

```python
    MAX_RETRIES_EXCEEDED = "max_retries_exceeded"
    CANCELLED_BY_USER = "cancelled_by_user"
    UPSTREAM_FAILED = "upstream_failed"

@dataclass
class TaskDefinition:
    """Definition of a single task within a pipeline."""
    id: str
    name: str
    type: str  # "extract", "transform", "load"
    config: Dict[str, Any]
    dependencies: List[str] = field(default_factory=list)
    retry_policy: RetryPolicy = field(default_factory=RetryPolicy)
    timeout_seconds: int = 3600

@dataclass
class PipelineDefinition:
    """Complete definition of an ETL pipeline."""
    id: str
    name: str
    description: str
    schedule: str  # cron expression
    tasks: List[TaskDefinition]
    parameters: Dict[str, Any] = field(default_factory=dict)
    created_at: datetime = field(default_factory=datetime.now)
    version: int = 1

@dataclass
class TaskExecution:
    """Runtime execution instance of a task."""
    task_id: str
    pipeline_run_id: str
    state: TaskState = TaskState.PENDING
    attempt_count: int = 0
```

```
    started_at: Optional[datetime] = None

    completed_at: Optional[datetime] = None

    error_message: Optional[str] = None

    logs: List[str] = field(default_factory=list)

    metrics: Dict[str, float] = field(default_factory=dict)



    def __post_init__(self):

        if not hasattr(self, 'execution_id'):

            self.execution_id = str(uuid.uuid4())
```

State Machine Implementation (complete):

PYTHON

```python
# src/etl/core/state_machine.py

from typing import Dict, Set, Optional, Tuple

from .models import TaskState, TaskEvent, TaskExecution


# Valid state transitions - defines the complete state machine

TRANSITIONS: Dict[TaskState, Dict[TaskEvent, TaskState]] = {

    TaskState.PENDING: {

        TaskEvent.DEPENDENCIES_MET: TaskState.QUEUED,

        TaskEvent.UPSTREAM_FAILED: TaskState.SKIPPED,

        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED,

    },

    TaskState.WAITING: {

        TaskEvent.DEPENDENCIES_MET: TaskState.QUEUED,

        TaskEvent.UPSTREAM_FAILED: TaskState.SKIPPED,

        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED,

    },

    TaskState.QUEUED: {

        TaskEvent.EXECUTION_STARTED: TaskState.RUNNING,

        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED,

    },

    TaskState.RUNNING: {

        TaskEvent.EXECUTION_COMPLETED: TaskState.SUCCESS,

        TaskEvent.EXECUTION_FAILED: TaskState.FAILED,

        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED,

    },

    TaskState.FAILED: {

        TaskEvent.RETRY_SCHEDULED: TaskState.RETRYING,

        TaskEvent.MAX_RETRIES_EXCEEDED: TaskState.FAILED,  # terminal state

    },

    TaskState.RETRYING: {

        TaskEvent.EXECUTION_STARTED: TaskState.RUNNING,

        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED,

    },
```

```
    # Terminal states have no outgoing transitions
    TaskState.SUCCESS: {},
    TaskState.CANCELLED: {},
    TaskState.SKIPPED: {},
}

def transition(execution: TaskExecution, event: TaskEvent, error_message: Optional[str] = None) -> bool:
    """
    Attempt to transition task execution to new state based on event.
    Returns True if transition was valid and applied, False otherwise.
    """
    current_state = execution.state

    if current_state not in TRANSITIONS:
        return False

    valid_events = TRANSITIONS[current_state]
    if event not in valid_events:
        return False

    # Apply the transition
    new_state = valid_events[event]
    execution.state = new_state

    # Update execution metadata based on transition
    if event == TaskEvent.EXECUTION_STARTED:
        execution.started_at = datetime.now()
        execution.attempt_count += 1
    elif event == TaskEvent.EXECUTION_COMPLETED:
        execution.completed_at = datetime.now()
    elif event == TaskEvent.EXECUTION_FAILED:
        execution.completed_at = datetime.now()
        if error_message:
```

```
            execution.error_message = error_message


    return True


def get_terminal_states() -> Set[TaskState]:
    """Return set of states that have no outgoing transitions."""
    return {state for state, transitions in TRANSITIONS.items() if not transitions}


def is_terminal_state(state: TaskState) -> bool:
    """Check if a state is terminal (has no outgoing transitions)."""
    return state in get_terminal_states()
```

Core Logic Skeleton

DAG Engine Interface (skeleton for student implementation):

PYTHON

```python
# src/etl/dag/engine.py

from typing import Optional, List, Dict, Set

from ..core.models import PipelineDefinition, TaskDefinition


class DAGEngine:

    """Core engine for pipeline definition management and validation."""


    def __init__(self, pipeline_storage_path: str):

        """Initialize DAG engine with pipeline storage location."""

        self.storage_path = pipeline_storage_path

        self._pipeline_cache: Dict[str, PipelineDefinition] = {}


    def get_pipeline(self, pipeline_id: str) -> Optional[PipelineDefinition]:

        """

        Retrieve pipeline definition by ID.

        Returns None if pipeline doesn't exist.

        """

        # TODO 1: Check local cache first - return cached version if available

        # TODO 2: Load pipeline from storage (YAML file) if not in cache

        # TODO 3: Parse YAML content into PipelineDefinition object

        # TODO 4: Store in cache for future requests

        # TODO 5: Return PipelineDefinition or None if file doesn't exist

        # Hint: Use self.storage_path + pipeline_id + ".yaml" as file path

        pass


    def validate_pipeline(self, pipeline: PipelineDefinition) -> List[str]:

        """

        Validate pipeline definition and return list of error messages.

        Returns empty list if pipeline is valid.

        """

        errors = []

        # TODO 1: Check that all task IDs are unique within pipeline

        # TODO 2: Validate that all task dependencies reference existing tasks
```

```
        # TODO 3: Check for circular dependencies using cycle detection
        # TODO 4: Validate that each task has required config fields for its type
        # TODO 5: Check that cron schedule expression is valid
        # Hint: Use self._detect_cycles() helper method for dependency validation
        return errors

    def _detect_cycles(self, tasks: List[TaskDefinition]) -> bool:
        """
        Detect if task dependencies contain cycles using depth-first search.
        Returns True if cycles are found, False otherwise.
        """
        # TODO 1: Build adjacency list from task dependencies
        # TODO 2: Track visited nodes and recursion stack
        # TODO 3: For each unvisited task, start DFS traversal
        # TODO 4: If we encounter a node already in recursion stack, cycle detected
        # TODO 5: Return True if any cycle found, False otherwise
        # Hint: Use three colors (white=unvisited, gray=visiting, black=done)
        pass

    def get_execution_order(self, pipeline: PipelineDefinition) -> List[List[str]]:
        """
        Return task IDs grouped by execution level using topological sort.
        Tasks in same inner list can execute in parallel.
        """
        # TODO 1: Build dependency graph with in-degree counting
        # TODO 2: Start with tasks that have zero dependencies (in-degree = 0)
        # TODO 3: Process tasks level by level, removing edges as we go
        # TODO 4: Add tasks to next level when their in-degree reaches 0
        # TODO 5: Return list of execution levels for parallel scheduling
        # Hint: Use queue for breadth-first processing of each level
        pass
```

Language-Specific Hints

Python Specific Recommendations:

- Use `dataclasses` with `frozen=True` for immutable data structures like `PipelineDefinition`
- Implement `__post_init__` in dataclasses for validation and computed fields
- Use `typing.Protocol` for defining connector interfaces that can be implemented by third parties
- Use `concurrent.futures.ThreadPoolExecutor` for parallel task execution with configurable pool size
- Use `pathlib.Path` for all file system operations instead of string manipulation
- Use `logging` module with structured logging (JSON format) for production deployments
- Use `yaml.safe_load()` for parsing pipeline definitions to prevent code injection
- Implement connection pooling using `queue.Queue` for database connectors

Error Handling Patterns:

```python
# Custom exception hierarchy for better error handling

class ETLError(Exception):

    """Base exception for all ETL system errors."""

    pass


class PipelineValidationError(ETLError):

    """Raised when pipeline definition is invalid."""

    def __init__(self, pipeline_id: str, errors: List[str]):

        self.pipeline_id = pipeline_id

        self.errors = errors

        super().__init__(f"Pipeline {pipeline_id} validation failed: {errors}")


class TaskExecutionError(ETLError):

    """Raised when task execution fails."""

    def __init__(self, task_id: str, attempt: int, cause: Exception):

        self.task_id = task_id

        self.attempt = attempt

        self.cause = cause

        super().__init__(f"Task {task_id} failed on attempt {attempt}: {cause}")
```

Milestone Checkpoints

After implementing core data models and state machine:

1. Run unit tests: `python -m pytest tests/unit/test_models.py -v`
2. Expected output: All state transition tests pass, invalid transitions properly rejected
3. Manual verification: Create a `TaskExecution` instance and call `transition()` with various events
4. Check that terminal states (SUCCESS, FAILED, CANCELLED) don't accept any transitions
5. Verify that attempt_count increments only on EXECUTION_STARTED events

After implementing DAG Engine basic functionality:

1. Create a sample pipeline YAML file with 3-4 tasks and dependencies
2. Run: `python -c "from etl.dag.engine import DAGEngine; engine = DAGEngine('pipelines/'); print(engine.get_pipeline('test'))"`
3. Expected behavior: Pipeline loads successfully and prints PipelineDefinition object
4. Test validation: Create pipeline with circular dependency, verify `validate_pipeline()` catches it
5. Test execution order: Verify `get_execution_order()` returns correct parallel execution groups

After implementing basic component communication:

1. Start scheduler in one terminal: `python scripts/start_scheduler.py`
2. Start executor in another terminal: `python scripts/start_executor.py`
3. Submit a simple pipeline through REST API: `curl -X POST localhost:8080/pipelines/test/run`
4. Check that both components log the pipeline execution flow
5. Verify that task state transitions are properly communicated between components

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Pipeline fails to load	YAML syntax error or missing file	Check file exists, validate YAML syntax	Fix YAML formatting, verify file permissions
Circular dependency error	Invalid task dependencies in pipeline definition	Draw dependency graph on paper	Remove circular references, add proper task ordering
Tasks stuck in PENDING state	Dependency resolution failure	Check task dependency configuration	Verify all dependencies reference existing tasks
High memory usage during execution	Large dataset loading without streaming	Monitor memory usage during connector operations	Implement streaming in connectors, reduce batch sizes
Components can't communicate	Network configuration or service discovery issues	Check port binding and firewall rules	Verify component startup order and configuration
State transitions fail	Invalid state machine transitions	Enable debug logging for state transitions	Check that events match current state in TRANSITIONS table

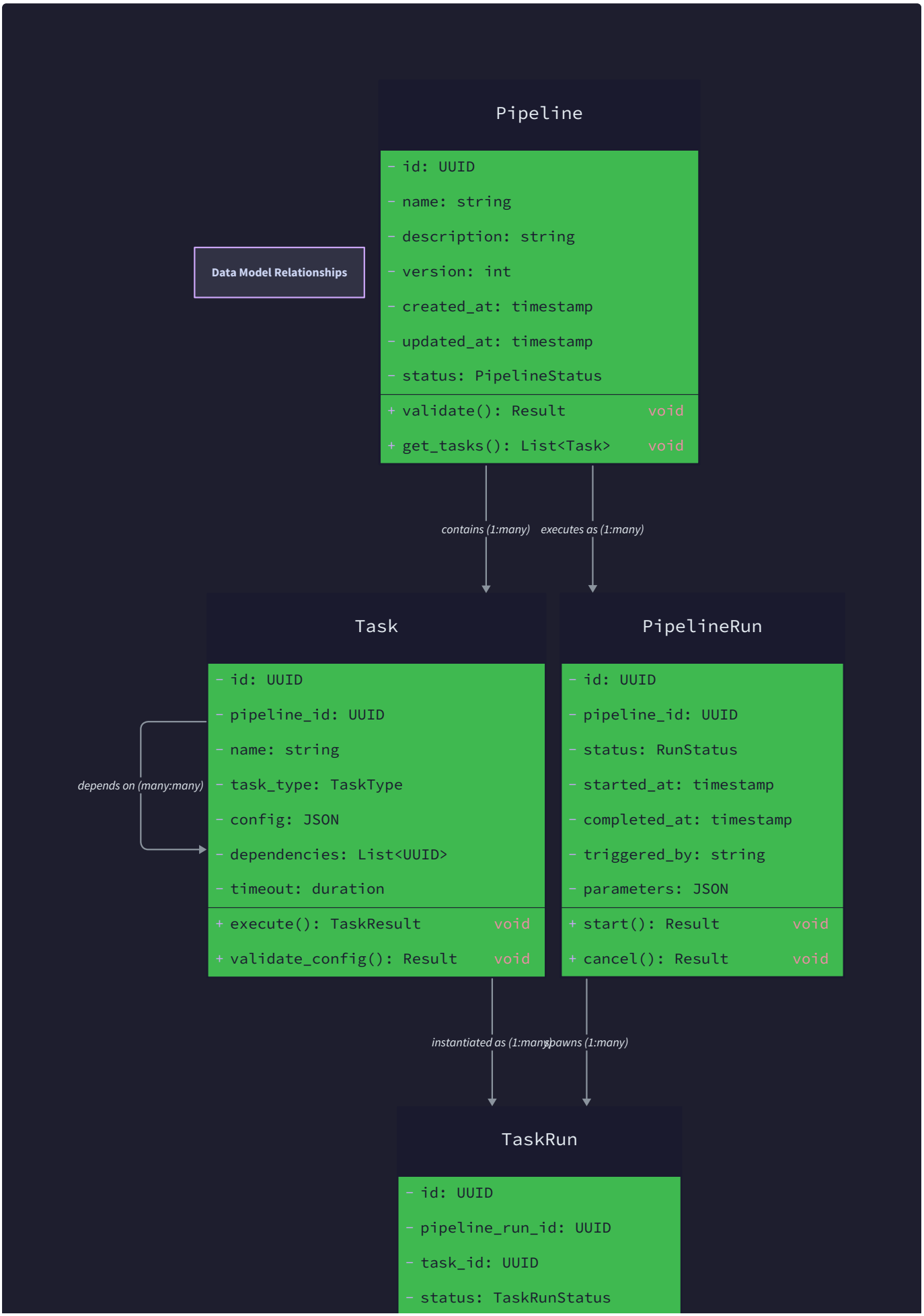
Data Model

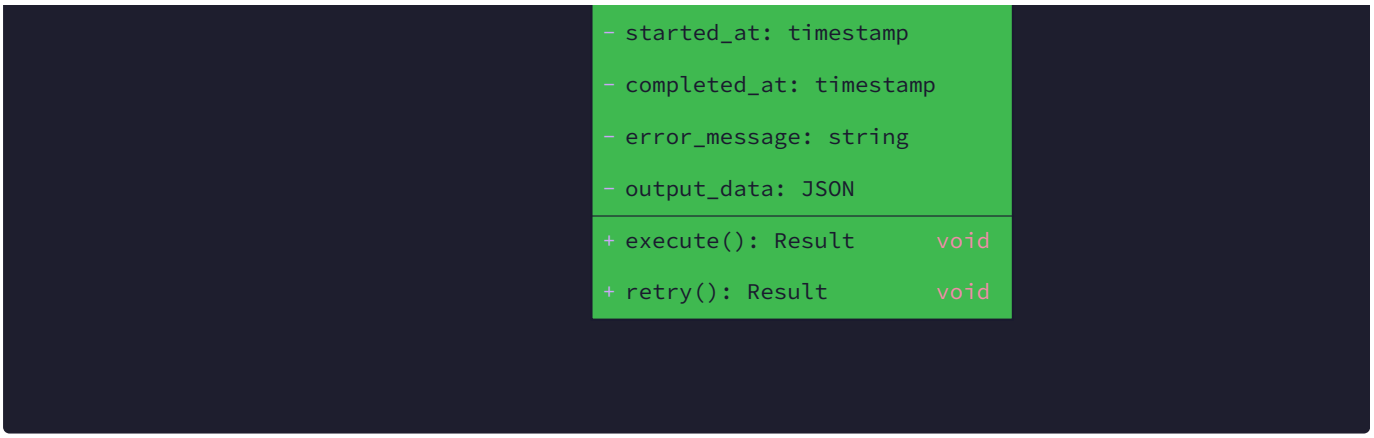
> **Milestone(s):** Foundation for all milestones - provides the data structures and state management that underpin pipeline definition (Milestone 1), data processing (Milestones 2-3), and orchestration (Milestone 4).

Mental Model: Digital Blueprint and Construction Log

Think of the data model as the combination of **architectural blueprints** and a **construction project log**. The pipeline and task definitions are like detailed blueprints that specify exactly what needs to be built, in what order, and with what materials. The runtime state model is like the foreman's log that tracks which workers are doing what, when each task started and finished, and any problems encountered. The metadata and lineage system is like a comprehensive project history that documents every decision, every change order, and the complete chain of how raw materials became the finished building.

Just as you wouldn't start construction without blueprints, and you wouldn't manage a complex construction project without tracking progress and maintaining detailed records, an ETL system needs these three layers of data organization to function reliably at scale.


Data Model Relationships
Pipeline
- id: UUID
- name: string
- description: string
- version: int
- created_at: timestamp
- updated_at: timestamp
- status: PipelineStatus
+ validate(): Result void
+ get_tasks(): List<Task> void
contains (1:many)
executes as (1:many)
Task
- id: UUID
- pipeline_id: UUID
- name: string
- task_type: TaskType
- config: JSON
- dependencies: List<UUID>
- timeout: duration
+ execute(): TaskResult void
+ validate_config(): Result void
depends on (many:many)
PipelineRun
- id: UUID
- pipeline_id: UUID
- status: RunStatus
- started_at: timestamp
- completed_at: timestamp
- triggered_by: string
- parameters: JSON
+ start(): Result void
+ cancel(): Result void
instantiated as (1:many)
spawns (1:many)
TaskRun
- id: UUID
- pipeline_run_id: UUID
- task_id: UUID
- status: TaskRunStatus



The data model serves as the foundation for all system operations, providing the structured representation of pipeline configurations, execution state, and historical metadata. This model must support concurrent access patterns, handle state transitions safely, and maintain data integrity across distributed components.

Pipeline and Task Definitions

The pipeline definition schema establishes the static blueprint for ETL workflows, capturing the declarative specification of what should happen without concern for runtime execution details. This separation between definition and execution enables versioning, validation, and reuse of pipeline logic across different environments.

PipelineDefinition Structure

The `PipelineDefinition` serves as the top-level container for all pipeline metadata and task specifications. Each pipeline represents a complete ETL workflow with its own scheduling, parameters, and dependency graph.

Field	Type	Description
id	str	Unique identifier for the pipeline, used for referencing and scheduling
name	str	Human-readable display name for the pipeline
description	str	Detailed explanation of pipeline purpose and business logic
schedule	str	Cron expression or event trigger specification for pipeline execution
tasks	List[TaskDefinition]	Complete list of all tasks that comprise this pipeline
parameters	dict	Default parameter values that can be overridden at runtime
created_at	datetime	Timestamp when this pipeline version was first created
version	int	Monotonically increasing version number for schema evolution

The pipeline identifier must be globally unique within the system and should follow a hierarchical naming convention that reflects organizational structure and purpose. The version field enables schema evolution and rollback capabilities, while the parameters dictionary provides a mechanism for runtime customization without modifying the core pipeline definition.

> **Critical Design Insight**: Pipeline definitions are immutable once created. Any modification creates a new version, ensuring that running pipelines complete with their original logic while new runs use updated definitions.

TaskDefinition Structure

Individual tasks represent atomic units of work within the pipeline, each encapsulating a specific transformation, extraction, or loading operation. Tasks must be designed to be idempotent and independently executable to support retry and recovery scenarios.

Field	Type	Description
id	str	Unique task identifier within the pipeline scope
name	str	Human-readable task name for monitoring and debugging
type	str	Task type identifier that determines execution behavior
config	dict	Task-specific configuration parameters and connection details
dependencies	List[str]	List of upstream task IDs that must complete before this task
retry_policy	RetryPolicy	Configuration for automatic retry behavior on failure
timeout_seconds	int	Maximum execution time before task is considered failed

The task type field determines which executor will handle the task and defines the expected structure of the config dictionary. Common task types include `sql_transform`, `api_extract`, `file_load`, and `python_udf`. The dependencies list establishes the DAG structure and must be validated to ensure no cycles exist.

> **Architecture Decision: Task Configuration Flexibility**
>
> - **Context**: Tasks need varying configuration parameters based on their type, but we want type safety and validation
> - **Options Considered**:
>   1. Strongly typed task subclasses with specific fields
>   2. Generic config dictionary with runtime validation
>   3. JSON schema validation for each task type
> - **Decision**: Generic config dictionary with pluggable validation per task type
> - **Rationale**: Provides maximum flexibility for custom task types while maintaining validation. Easier to extend than rigid inheritance hierarchies.
> - **Consequences**: Requires runtime validation and clear documentation of expected config schemas per task type

Option	Pros	Cons
Strongly typed subclasses	Compile-time safety, clear API	Rigid, hard to extend, complex inheritance
Generic config dict	Flexible, easy custom tasks	Runtime validation only, potential config errors
JSON schema validation	Good balance, clear contracts	Additional schema maintenance overhead

RetryPolicy Configuration

Retry policies define how the system should respond to task failures, balancing between automatic recovery and avoiding infinite retry loops. The policy configuration must be expressive enough to handle different failure types with appropriate retry strategies.

Field	Type	Description
max_attempts	int	Total number of execution attempts including the initial run
backoff_seconds	int	Base delay between retry attempts
exponential_backoff	bool	Whether to double the delay after each failure
retry_on_error_types	List[str]	Specific error categories that should trigger retries

The retry policy enables sophisticated failure handling by distinguishing between transient errors (network timeouts, temporary resource unavailability) and permanent errors (authentication failures, malformed queries). The `retry_on_error_types` field allows fine-

grained control over which failures justify retry attempts.

Dependency Resolution and Validation

Pipeline definitions undergo comprehensive validation during registration to catch configuration errors early and ensure reliable execution. The validation process examines both individual task configurations and the overall pipeline structure.

Pipeline Validation Steps:

1. **Task ID Uniqueness**: Verify that all task IDs within a pipeline are unique and follow naming conventions
2. **Dependency Reference Validation**: Confirm that all task dependencies refer to valid task IDs within the same pipeline
3. **Cycle Detection**: Perform topological analysis to ensure the dependency graph forms a valid DAG with no cycles
4. **Task Configuration Validation**: Validate each task's config dictionary against its type-specific schema
5. **Resource Requirement Analysis**: Check that resource requirements don't exceed system capacity limits
6. **Parameter Substitution Verification**: Ensure all parameter references in task configs have corresponding defaults or required parameters

The `validate_pipeline(pipeline) -> List[str]` function performs this comprehensive validation and returns a list of human-readable error messages. An empty list indicates a valid pipeline ready for execution.

> **Common Pitfall**: Circular Dependencies
>
> ⚠️ **Pitfall: Hidden Circular Dependencies** Developers often create circular dependencies through indirect chains (A→B→C→A) that aren't obvious in large pipelines. The validation must detect these transitively, not just check immediate dependencies. Always run full cycle detection using depth-first search with a visited set to catch these subtle cycles.

Runtime State Model

The runtime state model tracks the dynamic execution of pipeline definitions, maintaining detailed state information for active runs, completed executions, and failed attempts. This model must support concurrent access from multiple system components while ensuring consistency and auditability.

Pipeline Run Lifecycle

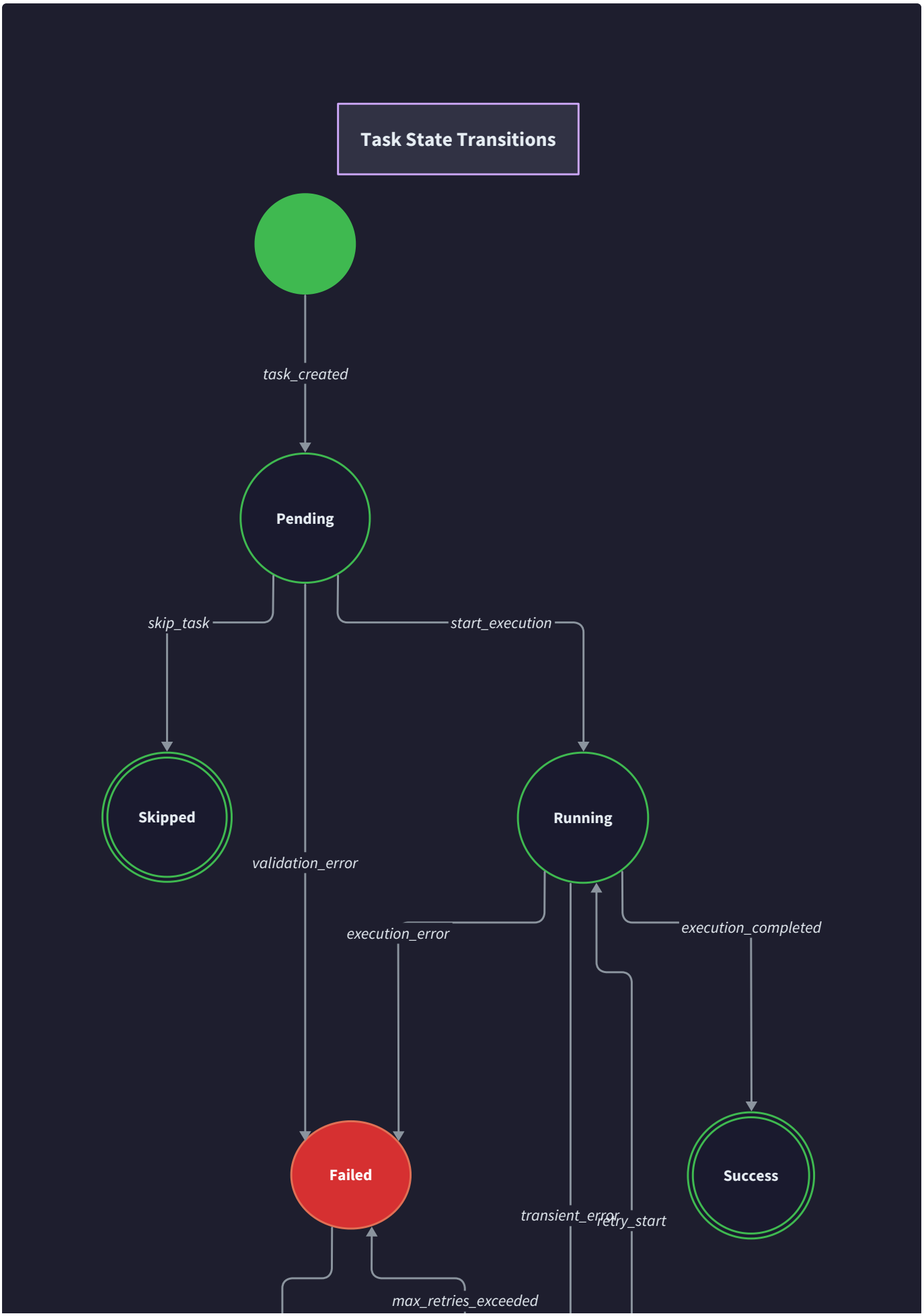
Each pipeline execution creates a `PipelineRun` instance that tracks the overall execution progress and coordinates individual task executions. Pipeline runs serve as the primary unit for monitoring, alerting, and historical analysis.

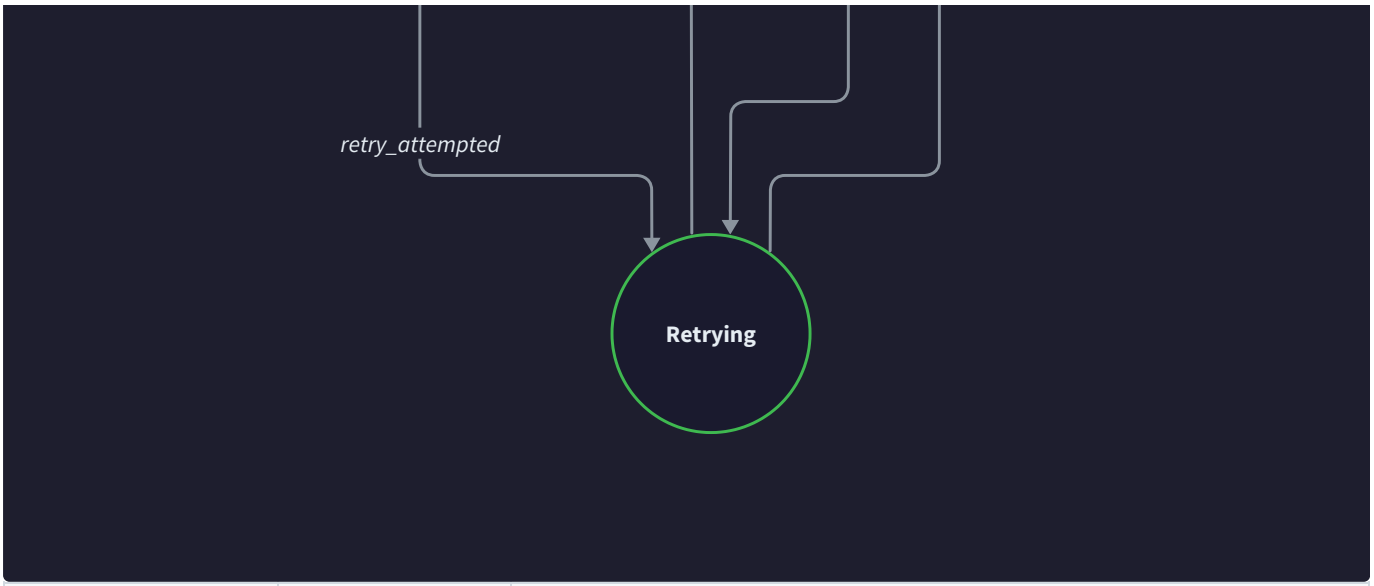
Field	Type	Description
run_id	str	Unique identifier for this specific pipeline execution
pipeline_id	str	Reference to the pipeline definition being executed
pipeline_version	int	Version of pipeline definition used for this run
triggered_by	str	Identifier of user, scheduler, or event that initiated the run
triggered_at	datetime	Timestamp when the run was initiated
started_at	datetime	Timestamp when first task began execution
completed_at	datetime	Timestamp when all tasks finished (success or failure)
state	PipelineRunState	Overall run state (PENDING, RUNNING, SUCCESS, FAILED, CANCELLED)
parameters	dict	Runtime parameters used for this execution
task_executions	List[TaskExecution]	All task execution instances for this run

The pipeline run maintains the execution context and provides a consistent view of progress across all constituent tasks. The `pipeline_version` field ensures that runs can be analyzed against the correct pipeline definition even after newer versions are deployed.

Task Execution State Management

Individual task executions track the detailed progress of each task within a pipeline run. The state model must handle complex scenarios including retries, cancellations, and dependency failures while maintaining a clear audit trail.


Task State Transitions
task_created
Pending
skip_task
start_execution
Skipped
validation_error
Running
execution_error
execution_completed
Failed
Success
transient_error
retry_start
max_retries_exceeded



Field	Type	Description
task_id	str	Reference to task definition within the pipeline
pipeline_run_id	str	Reference to parent pipeline run
state	TaskState	Current execution state of this task
attempt_count	int	Number of execution attempts including current attempt
started_at	datetime	Timestamp when task execution began
completed_at	datetime	Timestamp when task finished (success or failure)
error_message	str	Detailed error description for failed executions
logs	List[str]	Chronological log entries from task execution
metrics	Dict[str,float]	Performance metrics (duration, records processed, etc.)

The `TaskExecution` model captures both the current state and the complete execution history for each task attempt. This granular tracking enables detailed debugging and performance analysis while supporting retry logic and failure recovery.

State Transition Logic

Task state transitions follow a deterministic state machine that ensures consistent behavior across different execution scenarios. The transition function `transition(execution, event, error_message) -> bool` attempts to move a task execution to the appropriate next state based on the triggering event.

Current State	Event	Next State	Action Taken
PENDING	DEPENDENCIES_MET	QUEUED	Add to execution queue
QUEUED	EXECUTION_STARTED	RUNNING	Update started_at timestamp
RUNNING	EXECUTION_COMPLETED	SUCCESS	Update completed_at, record metrics
RUNNING	EXECUTION_FAILED	FAILED or RETRYING	Check retry policy, increment attempt_count
RETRYING	EXECUTION_STARTED	RUNNING	Begin new execution attempt
RUNNING	CANCELLED_BY_USER	CANCELLED	Terminate execution, cleanup resources
WAITING	UPSTREAM_FAILED	SKIPPED	Mark as skipped due to dependency failure
FAILED	MAX_RETRIES_EXCEEDED	FAILED	Final failure state, no more retries

The state machine enforces valid transitions and prevents invalid state changes that could compromise system integrity. Each transition is atomic and logged for audit purposes.

> **Architecture Decision: State Transition Atomicity**
>
> - **Context**: State transitions must be atomic to prevent race conditions in concurrent execution environments
> - **Options Considered**:
>   1. Database transactions for each state change
>   2. In-memory locks with periodic persistence
>   3. Event sourcing with append-only log
> - **Decision**: Database transactions with optimistic locking for state changes
> - **Rationale**: Provides ACID guarantees while supporting distributed execution. Optimistic locking prevents contention in normal cases.
> - **Consequences**: Requires retry logic for concurrent updates, but ensures consistency and supports system recovery

Execution Metrics and Observability

The runtime state model captures comprehensive metrics to enable monitoring, alerting, and performance optimization. Metrics are collected at both the task and pipeline level to provide different granularities of visibility.

Task-Level Metrics:

Metric Name	Type	Description
execution_duration_seconds	float	Wall-clock time from start to completion
cpu_usage_percent	float	Average CPU utilization during execution
memory_usage_mb	float	Peak memory consumption during execution
records_processed	float	Number of data records processed by the task
bytes_processed	float	Total bytes read and written by the task
error_count	float	Number of recoverable errors encountered during execution

Pipeline-Level Metrics:

Metric Name	Type	Description
total_duration_seconds	float	End-to-end pipeline execution time
task_success_count	float	Number of tasks that completed successfully
task_failure_count	float	Number of tasks that failed after all retries
critical_path_duration	float	Duration of longest task dependency chain
parallelism_achieved	float	Average number of tasks running concurrently
data_freshness_hours	float	Age of the oldest input data processed

These metrics enable both real-time monitoring and historical analysis for capacity planning and performance optimization.

Metadata and Lineage

The metadata and lineage system maintains comprehensive provenance information that tracks how data flows through the system, enabling compliance, debugging, and impact analysis. This system captures both technical lineage (what transformations were applied) and business lineage (what business processes were affected).

Data Lineage Tracking

Data lineage captures the complete transformation history of data as it moves through ETL pipelines, creating an auditable trail that can answer questions like "where did this data come from?" and "what processes will be affected if this source changes?"

Dataset Registration:

Field	Type	Description
dataset_id	str	Unique identifier for the logical dataset
name	str	Human-readable dataset name
schema_version	int	Current schema version number
location	str	Physical location (table name, file path, etc.)
owner	str	Business owner responsible for dataset quality
tags	List[str]	Metadata tags for discovery and classification
created_at	datetime	When this dataset was first registered
last_updated	datetime	Most recent data refresh timestamp

Lineage Edge Tracking:

Field	Type	Description
source_dataset_id	str	Dataset that provides input data
target_dataset_id	str	Dataset that receives transformed data
pipeline_id	str	Pipeline that performs the transformation
task_id	str	Specific task within pipeline that creates the relationship
transformation_type	str	Type of transformation applied (JOIN, AGGREGATE, FILTER, etc.)
column_mappings	Dict[str,str]	Mapping from source columns to target columns
created_at	datetime	When this lineage relationship was established

The lineage system builds a directed graph of data dependencies that can be traversed in both directions to understand upstream sources and downstream consumers. This graph supports impact analysis when schema changes or data quality issues are detected.

Schema Evolution and Versioning

Schema evolution tracking ensures that changes to data structures are managed safely without breaking downstream consumers. The system maintains a complete history of schema changes and provides compatibility checking for pipeline modifications.

Schema Version History:

Field	Type	Description
dataset_id	str	Reference to the dataset being versioned
version	int	Monotonically increasing version number
schema_definition	dict	Complete schema specification (columns, types, constraints)
change_type	str	Type of change (ADD_COLUMN, DROP_COLUMN, CHANGE_TYPE, etc.)
change_description	str	Human-readable description of what changed
compatibility	str	Backward compatibility assessment (COMPATIBLE, BREAKING, UNKNOWN)
applied_by	str	User or system that applied this schema change
applied_at	datetime	Timestamp when change was applied

Schema Compatibility Rules:

1. **Backward Compatible Changes**: Adding optional columns, relaxing constraints, adding enum values
2. **Breaking Changes**: Removing columns, changing data types, adding required columns, tightening constraints
3. **Forward Compatible Changes**: Changes that older readers can safely ignore

The schema evolution system automatically analyzes proposed changes and identifies potentially affected downstream consumers, enabling proactive communication and migration planning.

Architecture Decision: Schema Evolution Strategy

- **Context**: Need to support schema changes without breaking existing pipelines while maintaining data quality
- **Options Considered**:
  1. Strict immutability - never change schemas
  2. Automatic migration with compatibility checking
  3. Explicit versioning with parallel schema support
- **Decision**: Explicit versioning with automated compatibility analysis and migration assistance
- **Rationale**: Balances safety with flexibility. Provides clear upgrade paths while catching breaking changes early.
- **Consequences**: Requires schema registry maintenance and coordination between teams for breaking changes

Audit Trail and Compliance

The audit trail maintains a complete record of all system activities for compliance, debugging, and security analysis. This includes user actions, system events, and data access patterns that might be required for regulatory compliance.

Audit Event Structure:

Field	Type	Description
event_id	str	Unique identifier for this audit event
event_type	str	Category of event (USER_ACTION, SYSTEM_EVENT, DATA_ACCESS)
actor	str	User, service, or system component that initiated the action
resource_type	str	Type of resource affected (PIPELINE, TASK, DATASET)
resource_id	str	Identifier of specific resource instance
action	str	Specific action performed (CREATE, UPDATE, DELETE, EXECUTE, READ)
timestamp	datetime	Precise timestamp when event occurred
ip_address	str	Network address of request origin
session_id	str	Session identifier for grouping related actions
details	dict	Action-specific details and parameters
result	str	Outcome of the action (SUCCESS, FAILURE, PARTIAL)

Common Audit Event Types:

Event Type	Actor	Action	Details Captured
Pipeline Execution	System	EXECUTE	Run parameters, duration, success/failure
Schema Change	User	UPDATE	Old/new schemas, compatibility assessment
Data Access	Pipeline/User	READ	Query executed, rows returned, data volume
Configuration Change	User	UPDATE	Before/after configuration, change reason
User Authentication	User	LOGIN/LOGOUT	Authentication method, success/failure reason

The audit trail supports compliance with regulations like GDPR, SOX, and industry-specific requirements by maintaining detailed records of who accessed what data when, and what changes were made to processing logic.

> **Critical Design Insight**: Audit events are write-only and immutable once created. They use append-only storage with cryptographic integrity checking to prevent tampering and ensure compliance requirements are met.

Data Quality and Validation History

The system tracks data quality metrics and validation results over time to identify trends, detect anomalies, and ensure data reliability for downstream consumers. This historical view enables proactive data quality management and root cause analysis.

Data Quality Metrics:

Field	Type	Description
dataset_id	str	Dataset being measured
pipeline_run_id	str	Pipeline run that generated these metrics
metric_name	str	Name of quality metric (COMPLETENESS, ACCURACY, CONSISTENCY)
metric_value	float	Measured value for this metric
threshold_min	float	Minimum acceptable value for this metric
threshold_max	float	Maximum acceptable value for this metric
status	str	Quality status (PASS, WARN, FAIL) based on thresholds
measured_at	datetime	When this measurement was taken
sample_size	int	Number of records included in measurement

Validation Rule Results:

Field	Type	Description
rule_id	str	Identifier for the validation rule
dataset_id	str	Dataset being validated
rule_type	str	Type of validation (NOT_NULL, RANGE_CHECK, FORMAT_VALIDATION)
rule_expression	str	Actual validation logic or SQL expression
records_checked	int	Total number of records evaluated
records_passed	int	Number of records that satisfied the rule
records_failed	int	Number of records that violated the rule
failure_examples	List[dict]	Sample records that failed validation
execution_time_ms	float	Time taken to execute this validation rule

This quality tracking enables automated alerting when data quality degrades and provides historical context for understanding data reliability trends over time.

⚠️ Pitfall: Metadata Storage Performance

Lineage and audit systems generate high-volume write workloads that can impact operational system performance. Design these systems with separate storage infrastructure and asynchronous processing to avoid slowing down pipeline execution. Consider using time-series databases or append-only storage optimized for write-heavy workloads.

Implementation Guidance

The data model implementation requires careful consideration of storage technologies, access patterns, and consistency requirements. The following guidance provides practical recommendations for implementing a production-ready system.

Technology Recommendations

Component	Simple Option	Advanced Option
Pipeline Definitions	JSON files + file system	PostgreSQL with versioning
Runtime State	SQLite with WAL mode	PostgreSQL with connection pooling
Metrics Storage	In-memory + periodic dumps	InfluxDB or TimescaleDB
Lineage Graph	PostgreSQL with recursive queries	Neo4j graph database
Audit Log	Structured log files	Elasticsearch with retention policies
Schema Registry	JSON Schema + file storage	Confluent Schema Registry

Recommended File Structure

```
etl_system/
├── models/
│   ├── __init__.py
│   ├── pipeline.py          # PipelineDefinition, TaskDefinition
│   ├── execution.py         # TaskExecution, PipelineRun
│   ├── lineage.py           # Dataset, LineageEdge
│   └── audit.py             # AuditEvent, QualityMetric
├── storage/
│   ├── __init__.py
│   ├── pipeline_store.py    # Pipeline CRUD operations
│   ├── execution_store.py   # Runtime state management
│   └── metadata_store.py    # Lineage and audit storage
├── validation/
│   ├── __init__.py
│   ├── pipeline_validator.py # Pipeline validation logic
│   └── schema_validator.py  # Schema compatibility checking
└── migrations/
    ├── 001_initial_schema.sql
    ├── 002_add_lineage.sql
    └── 003_audit_indices.sql
```

Infrastructure Starter Code

Database Schema Setup (Complete):

PYTHON

```python
# storage/schema.py

from sqlalchemy import create_engine, Column, String, Integer, DateTime, Float, JSON, Text, Boolean

from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy.orm import sessionmaker, relationship

from sqlalchemy.dialects.postgresql import UUID

import uuid

from datetime import datetime


Base = declarative_base()


class PipelineDefinitionModel(Base):

    __tablename__ = 'pipeline_definitions'



    id = Column(String, primary_key=True)

    name = Column(String, nullable=False)

    description = Column(Text)

    schedule = Column(String)

    tasks = Column(JSON, nullable=False)  # Serialized TaskDefinition list

    parameters = Column(JSON, default=dict)

    created_at = Column(DateTime, default=datetime.utcnow)

    version = Column(Integer, nullable=False, default=1)



    # Relationships

    pipeline_runs = relationship("PipelineRunModel", back_populates="pipeline")


class PipelineRunModel(Base):

    __tablename__ = 'pipeline_runs'



    run_id = Column(String, primary_key=True, default=lambda: str(uuid.uuid4()))

    pipeline_id = Column(String, nullable=False)

    pipeline_version = Column(Integer, nullable=False)

    triggered_by = Column(String, nullable=False)

    triggered_at = Column(DateTime, default=datetime.utcnow)

    started_at = Column(DateTime)
```

```
    completed_at = Column(DateTime)
    state = Column(String, nullable=False, default='PENDING')
    parameters = Column(JSON, default=dict)

    # Relationships
    pipeline = relationship("PipelineDefinitionModel", back_populates="pipeline_runs")
    task_executions = relationship("TaskExecutionModel", back_populates="pipeline_run")

class TaskExecutionModel(Base):
    __tablename__ = 'task_executions'

    execution_id = Column(String, primary_key=True, default=lambda: str(uuid.uuid4()))
    task_id = Column(String, nullable=False)
    pipeline_run_id = Column(String, nullable=False)
    state = Column(String, nullable=False, default='PENDING')
    attempt_count = Column(Integer, default=1)
    started_at = Column(DateTime)
    completed_at = Column(DateTime)
    error_message = Column(Text)
    logs = Column(JSON, default=list)  # List of log entries
    metrics = Column(JSON, default=dict)

    # Relationships
    pipeline_run = relationship("PipelineRunModel", back_populates="task_executions")

# Database connection and session management
def create_database_engine(connection_string: str):
    """Create SQLAlchemy engine with connection pooling."""
    engine = create_engine(
        connection_string,
        pool_size=20,
        max_overflow=30,
        pool_pre_ping=True,  # Verify connections before use
        echo=False  # Set to True for SQL debugging
```

```
    )
    return engine

def create_session_factory(engine):
    """Create session factory for database operations."""
    return sessionmaker(bind=engine, expire_on_commit=False)

def initialize_database(engine):
    """Create all tables and initial data."""
    Base.metadata.create_all(engine)
```

State Management Utilities (Complete):

PYTHON

```python
# models/state.py

from enum import Enum

from typing import Dict, Set, Optional

from dataclasses import dataclass


class TaskState(Enum):

    PENDING = "PENDING"

    WAITING = "WAITING"

    QUEUED = "QUEUED"

    RUNNING = "RUNNING"

    SUCCESS = "SUCCESS"

    FAILED = "FAILED"

    RETRYING = "RETRYING"

    CANCELLED = "CANCELLED"

    SKIPPED = "SKIPPED"


class TaskEvent(Enum):

    DEPENDENCIES_MET = "DEPENDENCIES_MET"

    EXECUTION_STARTED = "EXECUTION_STARTED"

    EXECUTION_COMPLETED = "EXECUTION_COMPLETED"

    EXECUTION_FAILED = "EXECUTION_FAILED"

    RETRY_SCHEDULED = "RETRY_SCHEDULED"

    MAX_RETRIES_EXCEEDED = "MAX_RETRIES_EXCEEDED"

    CANCELLED_BY_USER = "CANCELLED_BY_USER"

    UPSTREAM_FAILED = "UPSTREAM_FAILED"


# State transition mapping - defines valid transitions

TRANSITIONS: Dict[TaskState, Dict[TaskEvent, TaskState]] = {

    TaskState.PENDING: {

        TaskEvent.DEPENDENCIES_MET: TaskState.QUEUED,

        TaskEvent.UPSTREAM_FAILED: TaskState.SKIPPED,

        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED,

    },

    TaskState.WAITING: {
```

```
        TaskEvent.DEPENDENCIES_MET: TaskState.QUEUED,
        TaskEvent.UPSTREAM_FAILED: TaskState.SKIPPED,
        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED,
    },
    TaskState.QUEUED: {
        TaskEvent.EXECUTION_STARTED: TaskState.RUNNING,
        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED,
    },
    TaskState.RUNNING: {
        TaskEvent.EXECUTION_COMPLETED: TaskState.SUCCESS,
        TaskEvent.EXECUTION_FAILED: TaskState.FAILED,  # Will be modified by retry logic
        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED,
    },
    TaskState.RETRYING: {
        TaskEvent.EXECUTION_STARTED: TaskState.RUNNING,
        TaskEvent.MAX_RETRIES_EXCEEDED: TaskState.FAILED,
        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED,
    },
    # Terminal states have no outgoing transitions
    TaskState.SUCCESS: {},
    TaskState.FAILED: {},
    TaskState.CANCELLED: {},
    TaskState.SKIPPED: {},
}

def get_terminal_states() -> Set[TaskState]:
    """Return states that represent completed execution."""
    return {TaskState.SUCCESS, TaskState.FAILED, TaskState.CANCELLED, TaskState.SKIPPED}

def is_valid_transition(current_state: TaskState, event: TaskEvent) -> bool:
    """Check if a state transition is valid."""
    return event in TRANSITIONS.get(current_state, {})

def get_next_state(current_state: TaskState, event: TaskEvent) -> Optional[TaskState]:
```

```
    """Get the next state for a given current state and event."""

    return TRANSITIONS.get(current_state, {}).get(event)
```

Core Logic Skeleton Code

Pipeline Validation (Skeleton with TODOs):

PYTHON

```python
# validation/pipeline_validator.py

from typing import List, Set, Dict

from models.pipeline import PipelineDefinition, TaskDefinition


def validate_pipeline(pipeline: PipelineDefinition) -> List[str]:

    """

    Validate pipeline definition and return list of error messages.

    Empty list indicates valid pipeline.

    """

    errors = []



    # TODO 1: Validate pipeline-level fields (id, name, schedule format)

    # Hint: Use croniter library to validate cron expressions



    # TODO 2: Extract all task IDs and check for duplicates

    # Hint: Use set() to detect duplicates in task ID list



    # TODO 3: Validate each task definition individually

    # Hint: Call validate_task_definition for each task



    # TODO 4: Build dependency graph and check for cycles

    # Hint: Use depth-first search with visited/visiting sets



    # TODO 5: Verify all dependency references point to valid tasks

    # Hint: Check that every dependency ID exists in task_ids set



    # TODO 6: Check for unreachable tasks (tasks with no path from roots)

    # Hint: Find tasks with no dependencies, then traverse reachable tasks



    return errors


def validate_task_definition(task: TaskDefinition, valid_task_ids: Set[str]) -> List[str]:

    """Validate individual task definition."""

    errors = []
```

```
    # TODO 1: Check required fields are present and valid format
    # TODO 2: Validate task type is supported
    # TODO 3: Validate retry policy parameters are reasonable
    # TODO 4: Check timeout_seconds is positive
    # TODO 5: Validate task-specific configuration based on task type

    return errors

def detect_cycles(tasks: List[TaskDefinition]) -> List[str]:
    """Detect cycles in task dependency graph using DFS."""
    # TODO 1: Build adjacency list from task dependencies
    # TODO 2: Initialize visited and visiting sets for DFS
    # TODO 3: For each unvisited task, start DFS traversal
    # TODO 4: Mark nodes as visiting when entering, visited when exiting
    # TODO 5: If we encounter a visiting node, we found a cycle
    # TODO 6: Return detailed cycle information for debugging

    pass  # Implementation goes here

def find_unreachable_tasks(tasks: List[TaskDefinition]) -> List[str]:
    """Find tasks that cannot be reached from root tasks."""
    # TODO 1: Find root tasks (tasks with no dependencies)
    # TODO 2: Build adjacency list for forward traversal
    # TODO 3: Perform BFS/DFS from all root tasks
    # TODO 4: Mark all reachable tasks
    # TODO 5: Return list of unreachable task IDs

    pass  # Implementation goes here
```

State Transition Logic (Skeleton with TODOs):

PYTHON

```python
# models/execution.py

from typing import Optional

from datetime import datetime

from models.state import TaskState, TaskEvent, TRANSITIONS


def transition(execution: 'TaskExecution', event: TaskEvent, error_message: Optional[str] = None) -> bool:

    """

    Attempt state transition based on event. Returns True if transition succeeded.

    Updates execution object in-place if successful.

    """

    current_state = TaskState(execution.state)



    # TODO 1: Check if transition is valid using TRANSITIONS mapping

    # Hint: Use is_valid_transition helper function



    # TODO 2: Handle retry logic for EXECUTION_FAILED event

    # Check if task has remaining retry attempts and error type is retryable

    # Modify target state to RETRYING instead of FAILED if retry is appropriate



    # TODO 3: Update execution object with new state

    # Set appropriate timestamps (started_at, completed_at)

    # Update error_message if provided

    # Increment attempt_count for retries



    # TODO 4: Log the state transition for audit trail

    # Include old state, new state, event, and timestamp



    # TODO 5: Return success/failure based on whether transition was applied



    pass  # Implementation goes here

def should_retry(execution: 'TaskExecution', error_message: str) -> bool:

    """Determine if task execution should be retried based on retry policy."""

    # TODO 1: Check if attempt_count < retry_policy.max_attempts
```

```
    # TODO 2: Check if error type matches retry_on_error_types
    # TODO 3: Return boolean indicating if retry should happen

    pass  # Implementation goes here

def calculate_retry_delay(execution: 'TaskExecution') -> int:
    """Calculate delay in seconds before next retry attempt."""
    # TODO 1: Get base delay from retry_policy.backoff_seconds
    # TODO 2: Apply exponential backoff if retry_policy.exponential_backoff is True
    # TODO 3: Add jitter to prevent thundering herd (random ±20%)
    # TODO 4: Return delay in seconds

    pass  # Implementation goes here
```

Language-Specific Implementation Hints

Python-Specific Recommendations:

- Use `SQLAlchemy` with `alembic` for database schema migrations and ORM mapping
- Use `pydantic` for data validation and serialization with automatic type checking
- Use `enum.Enum` for state definitions to ensure type safety and IDE support
- Use `dataclasses` or `pydantic.BaseModel` for data transfer objects between components
- Use `asyncio` and `asyncpg` for high-performance async database operations if needed
- Use `structlog` for structured logging with consistent field names and JSON output
- Use `pytest` with `pytest-asyncio` for comprehensive testing including async code paths

Database Optimization Tips:

- Create indexes on frequently queried fields: `pipeline_run_id`, `state`, `started_at`
- Use database constraints to enforce data integrity (foreign keys, check constraints)
- Consider partitioning large tables by time (`completed_at`) for better query performance
- Use connection pooling to handle concurrent database access efficiently
- Implement database health checks and circuit breakers for resilience

Concurrency Considerations:

- Use optimistic locking for state transitions to handle concurrent updates safely
- Implement database transactions for operations that must be atomic
- Consider using database-level advisory locks for critical sections
- Design for idempotency - operations should be safe to retry

Milestone Checkpoints

Checkpoint 1: Basic Data Model (After Pipeline Definition Implementation)

- **Test Command**: `python -m pytest tests/test_models.py -v`
- **Expected Output**: All basic model creation and validation tests pass

- **Manual Verification**:
  1. Create a simple pipeline definition with 3 tasks
  2. Verify cycle detection catches circular dependencies
  3. Confirm validation rejects invalid cron expressions
- **Success Criteria**: Pipeline definitions can be created, validated, and stored without errors

Checkpoint 2: State Management (After Runtime State Implementation)

- **Test Command**: `python -m pytest tests/test_state_machine.py -v`
- **Expected Output**: All state transition tests pass, invalid transitions are rejected
- **Manual Verification**:
  1. Create task execution and verify initial PENDING state
  2. Trigger valid transitions and confirm state changes
  3. Attempt invalid transitions and verify they're rejected
- **Success Criteria**: State machine enforces valid transitions and tracks execution history

Checkpoint 3: Metadata Integration (After Lineage Implementation)

- **Test Command**: `python -m pytest tests/test_lineage.py -v`
- **Expected Output**: Lineage tracking and schema evolution tests pass
- **Manual Verification**:
  1. Execute pipeline and verify lineage relationships are created
  2. Query upstream and downstream datasets for a given dataset
  3. Test schema evolution with compatible and breaking changes
- **Success Criteria**: Complete data provenance tracking with schema evolution support

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Pipeline validation hangs	Circular dependency in large DAG	Add logging to cycle detection algorithm, check task dependency chains	Implement cycle detection with proper visited tracking
State transitions fail silently	Missing error handling in transition logic	Add logging to transition function, check TRANSITIONS mapping	Ensure all valid transitions are defined in state machine
Database deadlocks during concurrent runs	Multiple transactions updating same records	Enable database deadlock logging, identify conflicting queries	Use consistent lock ordering, shorter transactions
Memory usage grows over time	Task execution objects not garbage collected	Profile memory usage, check for circular references	Implement proper cleanup after task completion
Lineage queries are slow	Missing database indexes on relationship tables	Use EXPLAIN ANALYZE on slow queries	Add indexes on source_dataset_id, target_dataset_id
Schema validation errors unclear	Generic error messages in validation code	Add detailed error context with field names and values	Include specific field validation errors with suggested fixes

DAG Definition and Validation Engine

> **Milestone(s):** Milestone 1 - Pipeline DAG Definition: Implements pipeline definition with dependencies as directed acyclic graph, including DAG parsing, validation, cycle detection, and visualization capabilities.

Mental Model: Recipe Dependencies

Think of an ETL pipeline like preparing a complex multi-course dinner where dishes have strict preparation dependencies. Just as you cannot plate the main course before cooking it, or start cooking before chopping ingredients, ETL tasks must execute in a precise order based on their dependencies.

Consider making a Thanksgiving dinner: you must thaw the turkey before seasoning it, season it before cooking it, cook it before carving it, and carve it before serving. Meanwhile, side dishes like mashed potatoes can be prepared in parallel with the turkey cooking, but they depend on earlier steps like peeling and boiling potatoes. The cranberry sauce is completely independent and can be made at any time.

This cooking analogy maps perfectly to ETL pipelines. Each recipe step is a **task** with specific inputs (dependencies) and outputs (results). The overall meal plan is the **pipeline definition**. Some tasks can run in parallel (like cooking turkey and preparing vegetables simultaneously), while others must wait for their dependencies (you cannot make gravy until the turkey drippings are ready). The critical insight is that we need to validate the entire recipe before we start cooking to ensure we never encounter impossible situations like circular dependencies (needing the gravy to cook the turkey, but needing the turkey to make the gravy).

The **DAG Definition and Validation Engine** serves as both the head chef who designs the meal plan and the kitchen manager who ensures the recipe is logically sound before any cooking begins. It parses recipe definitions (YAML/Python configs), validates that all dependencies make sense (cycle detection), determines the optimal preparation order (topological sorting), and creates visual cooking schedules (DAG visualization) that kitchen staff can follow.

DAG Parsing and Validation

The DAG parsing and validation system transforms human-readable pipeline definitions into validated execution graphs that the orchestration engine can safely execute. This process involves multiple stages of parsing, validation, and graph construction, each with specific responsibilities and error handling requirements.

Pipeline Definition Parsing

The system supports two primary formats for pipeline definitions: YAML configuration files and Python-based definitions. YAML provides a declarative approach suitable for data engineers who prefer configuration-over-code, while Python definitions offer programmatic flexibility for complex dynamic pipelines.

The parsing engine first loads the raw configuration and performs structural validation to ensure all required fields are present and properly typed. The parser constructs `PipelineDefinition` objects that contain all necessary metadata and task specifications.

Field	Type	Description
id	str	Unique identifier for the pipeline across the entire system
name	str	Human-readable display name for UI and logging purposes
description	str	Detailed explanation of pipeline purpose and data flow
schedule	str	Cron expression or event trigger specification for execution timing
tasks	List[TaskDefinition]	Complete list of all tasks in the pipeline with their configurations
parameters	dict	Global pipeline parameters that can be referenced by tasks at runtime
created_at	datetime	Pipeline creation timestamp for versioning and audit purposes
version	int	Pipeline version number for schema evolution and rollback support

Each `TaskDefinition` within the pipeline contains comprehensive metadata about individual task execution requirements and dependencies:

Field	Type	Description
id	str	Unique task identifier within the pipeline scope
name	str	Human-readable task name for monitoring and debugging
type	str	Task type indicating which executor should handle this task (extract, transform, load)
config	dict	Task-specific configuration parameters including connection details and operations
dependencies	List[str]	List of upstream task IDs that must complete successfully before this task runs
retry_policy	RetryPolicy	Configuration for failure handling and automatic retry behavior
timeout_seconds	int	Maximum execution time before the task is considered failed and terminated

The retry policy configuration provides fine-grained control over failure recovery behavior:

Field	Type	Description
max_attempts	int	Maximum number of execution attempts including the initial attempt
backoff_seconds	int	Base delay between retry attempts in seconds
exponential_backoff	bool	Whether to apply exponential backoff increasing delay between retries
retry_on_error_types	List[str]	Specific error types that should trigger retries vs immediate failure

> **Decision: Support Both YAML and Python Pipeline Definitions**
>
> - **Context**: Data engineers have varying preferences for configuration management - some prefer declarative YAML while others need programmatic Python flexibility
> - **Options Considered**: YAML-only (simple but limited), Python-only (flexible but complex), Both formats (flexible but more maintenance)
> - **Decision**: Support both YAML and Python definitions with a common internal representation
> - **Rationale**: YAML handles 80% of use cases with better readability, while Python enables complex dynamic pipelines and parameterization that configuration files cannot express
> - **Consequences**: Increases parser complexity but maximizes adoption across different user preferences and use cases

Dependency Graph Construction

After parsing individual task definitions, the engine constructs a directed graph representation where nodes represent tasks and edges represent dependencies. This graph construction process validates referential integrity by ensuring all declared dependencies reference existing tasks within the pipeline.

The graph construction algorithm follows these steps:

1. Create a node for each task definition using the task ID as the unique identifier
2. Iterate through each task's dependency list and verify that each referenced task ID exists in the pipeline
3. Create directed edges from dependency tasks to the current task (upstream → downstream)
4. Build adjacency lists for efficient graph traversal during validation and execution planning
5. Create reverse adjacency lists to support upstream impact analysis during failures

The resulting graph structure supports multiple query patterns required by the orchestration engine:

- **Forward traversal**: Finding all downstream tasks affected by a failure
- **Backward traversal**: Identifying all upstream dependencies that must complete before a task can run

- **Parallel task identification**: Finding tasks with satisfied dependencies that can execute simultaneously
- **Critical path analysis**: Determining the longest dependency chain that affects overall pipeline completion time

Cycle Detection Algorithm

Cycle detection is the most critical validation step, as cyclical dependencies would cause the pipeline to deadlock indefinitely. The engine implements depth-first search (DFS) with coloring to detect cycles efficiently in O(V + E) time complexity where V is tasks and E is dependencies.

The three-color DFS algorithm works as follows:

1. Initialize all tasks as WHITE (unvisited)
2. For each WHITE task, start a DFS traversal
3. Mark the current task as GRAY (currently being processed) when entering its DFS branch
4. Recursively visit all downstream tasks (dependencies of current task)
5. If we encounter a GRAY task during traversal, we have found a cycle
6. Mark the task as BLACK (completely processed) when finishing its DFS branch
7. Report any cycles found with the specific tasks involved for debugging

The coloring approach distinguishes between back edges (which indicate cycles) and forward/cross edges (which are harmless). This precision prevents false positives that simpler visited/unvisited algorithms might produce in complex graphs.

> The critical insight is that GRAY nodes represent the current DFS path stack. Finding a GRAY node means we have encountered a task that depends on itself through a chain of intermediate dependencies.

Validation Rule Engine

Beyond structural graph validation, the engine enforces semantic business rules that ensure pipeline correctness:

Validation Rule	Description	Error Handling
Task ID Uniqueness	No duplicate task IDs within a pipeline	Reject pipeline with specific duplicate IDs listed
Dependency Existence	All referenced dependencies must exist as tasks	Report missing task IDs and suggest similar names
Self-Dependency Prevention	Tasks cannot depend on themselves directly	Identify self-referential tasks and remove invalid dependencies
Orphaned Task Detection	All tasks except roots must have at least one path from a root	Warn about unreachable tasks that will never execute
Dangling Dependency Detection	Dependencies must reference valid task IDs	List invalid references with suggestions for correction
Resource Constraint Validation	Task resource requirements must not exceed system limits	Warn about tasks that may fail due to resource constraints

The validation engine returns a structured list of errors, warnings, and suggestions rather than failing fast on the first issue. This comprehensive feedback helps pipeline developers fix multiple issues simultaneously rather than discovering them one at a time through trial and error.

Parameter Substitution and Templating

Pipeline definitions support parameterization through template variables that are resolved at parse time or runtime. The templating system enables environment-specific configurations and dynamic pipeline behavior without duplicating pipeline definitions.

The parameter resolution process handles multiple scopes with clear precedence rules:

1. **Global pipeline parameters**: Defined at the pipeline level and available to all tasks
2. **Task-specific parameters**: Override global parameters for individual tasks
3. **Runtime parameters**: Provided when triggering pipeline execution to customize behavior
4. **Environment variables**: System environment variables available for configuration injection

Template syntax follows standard conventions using double curly braces: `{{parameter_name}}` with support for default values: `{{parameter_name|default_value}}`. The templating engine validates that all referenced parameters are defined and substitutes values before final validation occurs.

Configuration Schema Validation

The parsing engine enforces strict schema validation to catch configuration errors early in the development cycle. JSON Schema definitions specify the exact structure, types, and constraints for pipeline and task configurations.

Schema validation covers multiple levels:

- **Structural validation**: Required fields, correct types, valid enumerations
- **Cross-field validation**: Constraints that span multiple configuration fields
- **Business rule validation**: Domain-specific rules like valid cron expressions and connection string formats
- **Version compatibility**: Ensuring configurations are compatible with the current engine version

The validation system provides detailed error messages with field paths, expected vs. actual values, and suggested corrections. This comprehensive feedback reduces the debugging time for pipeline developers and prevents runtime failures due to misconfiguration.

Execution Order Resolution

Once the DAG passes validation, the execution order resolution system determines the optimal sequence for task execution while maximizing parallelism and respecting all dependency constraints. This process involves sophisticated graph algorithms that balance execution efficiency with resource utilization.

Topological Sorting Algorithm

The foundation of execution order resolution is topological sorting, which produces a linear ordering of tasks such that for every dependency relationship, the upstream task appears before the downstream task in the sequence. The engine implements Kahn's algorithm for its stability and intuitive behavior.

Kahn's algorithm operates through the following steps:

1. Calculate the **in-degree** (number of incoming dependencies) for each task
2. Initialize a queue with all tasks that have zero in-degree (no dependencies)
3. While the queue is not empty, remove a task from the queue and add it to the result sequence
4. For each downstream task dependent on the current task, decrement its in-degree
5. If decrementing causes a downstream task's in-degree to reach zero, add it to the queue
6. Continue until all tasks are processed or the queue becomes empty with tasks remaining

The algorithm naturally handles parallel execution opportunities by identifying tasks with zero in-degree at each step. Tasks removed from the queue simultaneously can execute in parallel since they have no mutual dependencies.

Parallel Execution Planning

Beyond basic topological ordering, the execution planner groups tasks into **execution levels** that maximize parallelism while respecting resource constraints. Each execution level contains tasks that can run simultaneously without violating dependencies.

The level-based execution planning algorithm works as follows:

1. Start with level 0 containing all tasks with no dependencies
2. For each subsequent level, include tasks whose dependencies are all satisfied by previous levels

3. Continue creating levels until all tasks are assigned
4. Within each level, apply resource-based scheduling to avoid oversubscription
5. Generate execution plan with explicit parallelism annotations

This approach provides several advantages over simple topological sorting:

- **Resource optimization**: Tasks within a level can be scheduled based on available CPU, memory, and I/O capacity
- **Failure isolation**: If a task fails, only subsequent levels containing dependent tasks need to be cancelled
- **Progress visualization**: Users can see pipeline progress as completion percentages within each level
- **Dynamic rescheduling**: Failed tasks can be retried without recalculating the entire execution plan

Critical Path Analysis

The execution planner performs critical path analysis to identify the longest dependency chain that determines minimum pipeline completion time. This analysis helps with resource allocation decisions and provides realistic completion time estimates.

Critical path calculation involves:

1. Calculate the **longest path** from each task to any terminal task (task with no downstream dependencies)
2. Identify tasks that lie on the critical path - any delay in these tasks delays the entire pipeline
3. Prioritize critical path tasks for resource allocation and monitoring
4. Provide completion time estimates based on critical path length and historical task durations

Tasks not on the critical path have **slack time** - the amount they can be delayed without affecting overall pipeline completion. The scheduler uses slack time information to optimize resource utilization and handle transient failures gracefully.

Dynamic Dependency Resolution

Some pipelines require dynamic dependency resolution where task dependencies are determined at runtime based on data characteristics or external conditions. The execution planner supports conditional dependencies through predicate evaluation.

Dynamic dependency types include:

Dependency Type	Description	Resolution Strategy
Conditional Dependencies	Dependencies that apply only when specific conditions are met	Evaluate predicates before adding edges to execution graph
Data-Driven Dependencies	Dependencies determined by upstream task output characteristics	Re-evaluate dependency graph after each task completion
External Dependencies	Dependencies on external systems or scheduled events	Poll external conditions before marking dependencies as satisfied
Parameterized Dependencies	Dependencies that vary based on pipeline parameters	Resolve parameter values before constructing dependency graph

The dynamic resolution system maintains the execution graph as a mutable structure that can be safely modified during pipeline execution while preserving consistency and preventing cycles.

Resource-Aware Scheduling

The execution planner incorporates resource constraints to prevent system overload and ensure stable pipeline execution. Resource-aware scheduling considers multiple constraint types:

Resource Type	Constraint	Scheduling Strategy
CPU Cores	Maximum concurrent CPU-intensive tasks	Queue CPU-bound tasks when core limit reached
Memory	Total memory allocation across running tasks	Delay memory-intensive tasks until sufficient memory available
Database Connections	Connection pool limits per database	Serialize tasks accessing the same database when pool exhausted
API Rate Limits	External API call quotas and rate limits	Space API-dependent tasks to respect rate limiting windows
File System I/O	Concurrent file operations and disk bandwidth	Balance file-intensive tasks across available storage devices

The scheduler maintains resource allocation tracking and updates availability as tasks start and complete. This real-time resource management prevents cascading failures due to resource exhaustion while maximizing system utilization.

Execution Plan Optimization

The final execution plan undergoes optimization to improve overall pipeline performance and resource efficiency. Optimization techniques include:

Task Batching: Small tasks with similar resource requirements are batched together to reduce orchestration overhead and improve resource locality.

Prefetching: The planner identifies opportunities to prefetch data or establish connections before tasks need them, reducing task startup latency.

Load Balancing: Tasks are distributed across available execution nodes to balance resource utilization and prevent hotspots.

Checkpoint Placement: Strategic checkpoints are inserted to enable efficient failure recovery without recomputing the entire execution plan.

The optimized execution plan includes detailed scheduling metadata that the orchestration engine uses for efficient task execution and resource management.

DAG Visualization

DAG visualization transforms the abstract dependency graph into intuitive visual representations that help pipeline developers understand, debug, and communicate about complex data workflows. The visualization system provides multiple rendering modes optimized for different use cases and audiences.

Graph Rendering Engine

The visualization engine converts the internal DAG representation into multiple output formats suitable for different consumption scenarios. The core rendering pipeline processes the validated DAG structure and applies layout algorithms to create readable visual representations.

The rendering process involves several stages:

1. **Node positioning**: Apply graph layout algorithms (hierarchical, force-directed, or circular) to determine optimal task placement
2. **Edge routing**: Calculate connection paths between tasks that minimize visual clutter and overlapping
3. **Visual styling**: Apply consistent colors, shapes, and annotations based on task types and states
4. **Interactive elements**: Add hover tooltips, click handlers, and zoom/pan capabilities for exploration
5. **Export generation**: Produce static images, interactive HTML, or embeddable SVG formats

Layout Algorithms

Different graph layout algorithms serve different visualization needs and user preferences:

Algorithm	Use Case	Advantages	Limitations
Hierarchical (Layered)	Sequential pipelines with clear stages	Shows execution order clearly, minimal edge crossings	Requires acyclic graphs, can be wide with many parallel tasks
Force-Directed	Complex interconnected pipelines	Handles arbitrary graph structures, visually appealing	May not show execution order clearly, can be unstable
Circular	Pipelines with hub-and-spoke patterns	Compact representation, good for radial dependencies	Difficult to read with many levels, edge crossings
Grid-Based	Pipelines with regular structure	Predictable layout, easy to align with external systems	Rigid, may waste space, doesn't adapt to graph structure

The visualization engine automatically selects the most appropriate algorithm based on graph characteristics but allows manual override for specific presentation needs.

> **Decision: Support Multiple Layout Algorithms with Automatic Selection**
>
> - **Context**: Different pipeline structures benefit from different visualization approaches, and users have varying preferences for graph layout
> - **Options Considered**: Single algorithm (simple but limited), User choice only (overwhelming for new users), Automatic with override (complexity but optimal UX)
> - **Decision**: Implement automatic algorithm selection with manual override capability
> - **Rationale**: Automatic selection provides good defaults for 90% of cases while expert users can choose specific algorithms for presentation or debugging needs
> - **Consequences**: Increases implementation complexity but dramatically improves user experience across different pipeline types

Interactive Features

Modern DAG visualization requires interactive capabilities that enable pipeline exploration and debugging workflows. The visualization system implements rich interactivity through web-based rendering with JavaScript integration.

Core interactive features include:

Node Inspection: Clicking on task nodes displays detailed information including configuration parameters, execution history, dependency relationships, and current status. The inspection panel shows both design-time configuration and runtime execution metrics.

Dependency Tracing: Users can highlight dependency chains by selecting a task and visualizing all upstream dependencies (what must complete before this task) or downstream impacts (what depends on this task). This tracing capability is essential for impact analysis during debugging.

Execution Replay: The visualization can animate historical pipeline executions by showing task state changes over time. This temporal visualization helps understand execution bottlenecks, failure propagation, and resource utilization patterns.

Real-time Updates: During live pipeline execution, the visualization updates task states in real-time through WebSocket connections to the monitoring system. Color coding and progress indicators provide immediate feedback on pipeline health.

Filtering and Search: Large pipelines benefit from filtering capabilities that hide or highlight specific task types, execution states, or dependency patterns. Search functionality enables quick navigation to specific tasks by name or configuration attributes.

Graph Navigation: Zoom, pan, and minimap features enable exploration of large complex pipelines that don't fit entirely on screen. The navigation system maintains context and provides orientation cues for user spatial awareness.

Visual Design System

Consistent visual design creates intuitive understanding across different pipelines and reduces cognitive load for users switching between projects. The design system defines standardized visual elements and their meanings.

Node Styling Convention:

Task Type	Shape	Color	Icon
Extract	Rectangle	Blue	Database/API symbol
Transform	Diamond	Green	Gear/Function symbol
Load	Rectangle	Orange	Target/Destination symbol
Validation	Hexagon	Purple	Check/Warning symbol
Notification	Circle	Yellow	Bell/Message symbol

Edge Styling Convention:

Dependency Type	Line Style	Color	Arrow Style
Data Dependency	Solid	Black	Standard arrow
Control Dependency	Dashed	Gray	Hollow arrow
Conditional Dependency	Dotted	Blue	Diamond arrow
External Dependency	Dash-dot	Red	Square arrow

State Indication System:

Task State	Border Color	Fill Pattern	Animation
PENDING	Gray	None	None
RUNNING	Blue	Diagonal stripes	Pulsing
SUCCESS	Green	Solid	None
FAILED	Red	Cross-hatch	None
RETRYING	Orange	Dots	Spinning
SKIPPED	Gray	Faded	None

Export and Integration Capabilities

The visualization system supports multiple export formats to integrate with documentation systems, monitoring dashboards, and presentation tools.

Static Export Formats:

- **SVG**: Vector graphics suitable for documentation and high-quality printing
- **PNG/JPEG**: Raster images for embedding in presentations and reports
- **PDF**: Multi-page layouts for large pipelines with detailed annotations

Interactive Export Formats:

- **HTML**: Self-contained interactive visualizations that work without server connectivity
- **Embedding codes**: JavaScript widgets for integration into custom dashboards and monitoring systems
- **API endpoints**: Real-time data feeds for custom visualization tools

Integration with External Tools:

The visualization engine provides REST API endpoints that external tools can query to retrieve graph data in standardized formats (GraphML, DOT, JSON). This integration capability enables pipeline visualization within existing workflow management tools and custom monitoring dashboards.

Performance Optimization for Large Pipelines

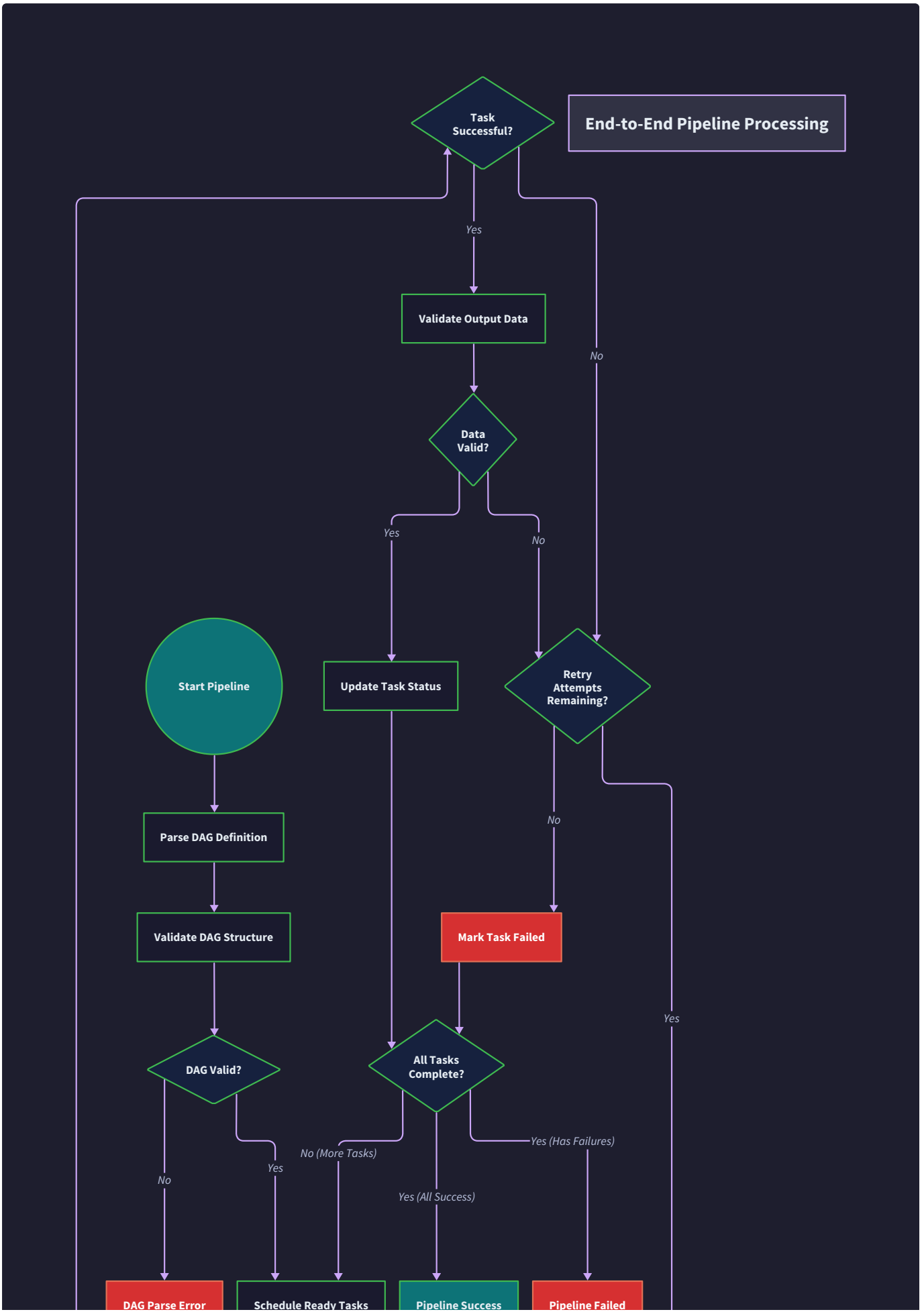
Large enterprise pipelines with hundreds or thousands of tasks require specialized rendering optimizations to maintain interactive performance. The visualization system implements several performance strategies:

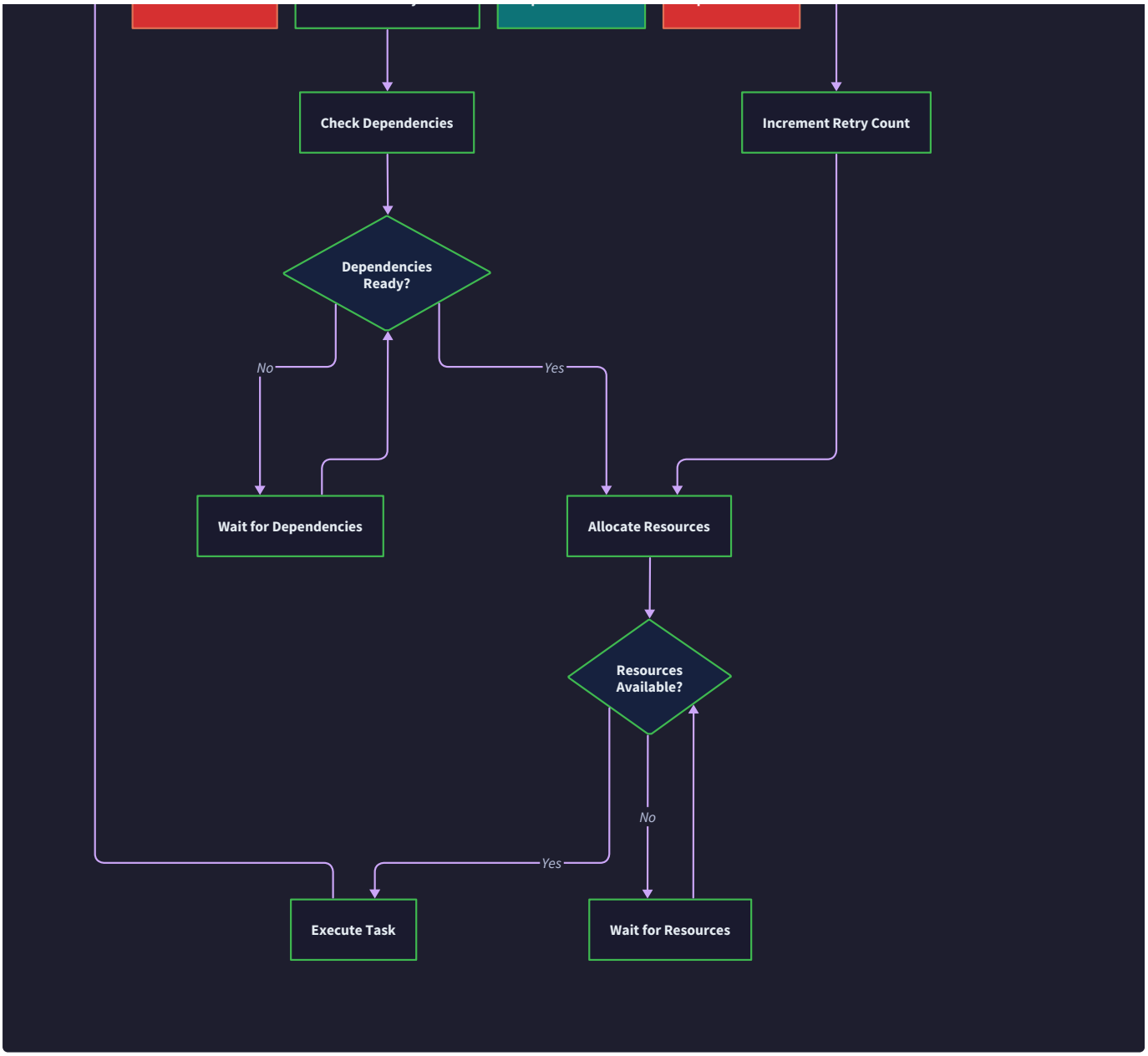
Level-of-Detail Rendering: Distant or zoomed-out nodes are rendered with simplified graphics while detailed rendering applies only to visible nodes. This technique maintains smooth interaction even with complex graphs.

Virtual Scrolling: Only visible portions of large graphs are rendered in the DOM while off-screen elements exist as lightweight data structures. This approach prevents browser performance degradation with massive pipelines.

Clustering and Summarization: Related tasks can be visually grouped into clusters with expand/collapse functionality. This hierarchical approach enables users to understand high-level pipeline structure while drilling into specific areas of interest.

Incremental Updates: Real-time state updates use efficient differential rendering that updates only changed elements rather than re-rendering the entire graph. This optimization maintains responsiveness during active pipeline execution.


End-to-End Pipeline Processing
Task Successful?
Yes
No
Validate Output Data
Data Valid?
Yes
No
Start Pipeline
Update Task Status
Retry Attempts Remaining?
Parse DAG Definition
No
Validate DAG Structure
Mark Task Failed
Yes
DAG Valid?
All Tasks Complete?
No
Yes
No (More Tasks)
Yes (Has Failures)
Yes (All Success)
DAG Parse Error
Schedule Ready Tasks
Pipeline Success
Pipeline Failed



Common Pitfalls in DAG Visualization

⚠️ **Pitfall: Over-complex Layout for Simple Pipelines** Many visualization systems apply sophisticated force-directed algorithms to simple sequential pipelines, resulting in unnecessarily complex layouts that obscure the straightforward execution flow. Simple pipelines benefit from hierarchical left-to-right layouts that clearly show the execution sequence. The solution is to analyze graph structure first and select layout algorithms appropriate for the specific pipeline topology.

⚠️ **Pitfall: Insufficient Visual Distinction Between Task Types** Using similar colors or shapes for different task types creates confusion and makes it difficult to understand pipeline structure at a glance. Users should be able to identify extract, transform, and load tasks immediately through visual cues. The solution is to establish clear visual conventions and apply them consistently across all pipeline visualizations.

⚠️ **Pitfall: Missing Real-time State Updates** Static visualizations that don't reflect current execution state force users to switch between monitoring tools and pipeline diagrams, reducing debugging efficiency. The visualization should integrate with the monitoring system to show current task states, progress indicators, and error conditions directly on the graph. This integration requires WebSocket connections or polling mechanisms to fetch real-time state updates.

⚠️ **Pitfall: Poor Performance with Large Graphs** Naive rendering implementations become unusably slow with pipelines containing hundreds of tasks, forcing users to avoid the visualization tool when they need it most. The solution involves implementing level-of-detail rendering, virtual scrolling, and clustering techniques to maintain interactive performance regardless of pipeline size.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Graph Processing	NetworkX (Python graphs)	igraph (high-performance graph algorithms)
YAML Parsing	PyYAML (standard library)	ruamel.yaml (preserves comments and formatting)
Visualization Backend	Matplotlib + NetworkX	Graphviz + PyGraphviz
Web Visualization	D3.js + SVG	Cytoscape.js (interactive graph library)
Schema Validation	jsonschema (Python)	Cerberus (more expressive validation rules)
Template Engine	Jinja2 (widely adopted)	Chevron (logic-less mustache templates)

B. Recommended File/Module Structure:

```
etl-pipeline/
  pipeline_engine/
    dag/
      __init__.py                    ← public API exports
      parser.py                      ← YAML/Python config parsing
      validator.py                   ← cycle detection and validation
      executor_planner.py            ← topological sort and execution planning
      visualizer.py                  ← graph rendering and layout
      models.py                      ← PipelineDefinition, TaskDefinition classes
      exceptions.py                  ← DAG-specific exception classes
    schemas/
      pipeline_schema.json           ← JSON schema for pipeline validation
      task_schemas/                  ← task type-specific schemas
        extract_schema.json
        transform_schema.json
        load_schema.json
    templates/
      visualization_template.html    ← HTML template for interactive graphs
    tests/
      test_parser.py
      test_validator.py
      test_execution_planner.py
      fixtures/                      ← sample pipeline definitions
        simple_pipeline.yaml
        complex_pipeline.yaml
        invalid_pipeline.yaml
```

C. Infrastructure Starter Code:

Pipeline Schema Validation (`schemas/pipeline_schema.json`):

```json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "required": ["id", "name", "tasks"],
  "properties": {
    "id": {
      "type": "string",
      "pattern": "^[a-zA-Z0-9_-]+$",
      "description": "Unique pipeline identifier"
    },
    "name": {
      "type": "string",
      "minLength": 1,
      "description": "Human-readable pipeline name"
    },
    "description": {
      "type": "string",
      "description": "Pipeline purpose and data flow description"
    },
    "schedule": {
      "type": "string",
      "description": "Cron expression or trigger specification"
    },
    "tasks": {
      "type": "array",
      "minItems": 1,
      "items": {"$ref": "#/definitions/task"}
    },
    "parameters": {
      "type": "object",
      "description": "Global pipeline parameters"
    }
  },
```

```
  "definitions": {
    "task": {
      "type": "object",
      "required": ["id", "name", "type", "config"],
      "properties": {
        "id": {
          "type": "string",
          "pattern": "^[a-zA-Z0-9_-]+$"
        },
        "name": {"type": "string", "minLength": 1},
        "type": {
          "type": "string",
          "enum": ["extract", "transform", "load", "validate", "notify"]
        },
        "config": {"type": "object"},
        "dependencies": {
          "type": "array",
          "items": {"type": "string"}
        },
        "timeout_seconds": {
          "type": "integer",
          "minimum": 1,
          "maximum": 86400
        }
      }
    }
  }
}
```

Graph Utility Functions (`dag/graph_utils.py`):

PYTHON

```python
from typing import Dict, List, Set, Tuple, Optional
from collections import defaultdict, deque
from enum import Enum

class NodeColor(Enum):
    WHITE = "white"  # Unvisited
    GRAY = "gray"    # Currently processing
    BLACK = "black"  # Completely processed

class CycleDetectionResult:
    def __init__(self, has_cycle: bool, cycle_path: List[str] = None):
        self.has_cycle = has_cycle
        self.cycle_path = cycle_path or []

def detect_cycles_dfs(adjacency_list: Dict[str, List[str]]) -> CycleDetectionResult:
    """
    Detect cycles in directed graph using DFS with three-color algorithm.
    Returns CycleDetectionResult with cycle information if found.
    """
    colors = {node: NodeColor.WHITE for node in adjacency_list}
    parent = {node: None for node in adjacency_list}

    def dfs_visit(node: str, path: List[str]) -> Optional[List[str]]:
        colors[node] = NodeColor.GRAY
        current_path = path + [node]

        for neighbor in adjacency_list.get(node, []):
            if colors[neighbor] == NodeColor.GRAY:
                # Found back edge - cycle detected
                cycle_start = current_path.index(neighbor)
                return current_path[cycle_start:] + [neighbor]
            elif colors[neighbor] == NodeColor.WHITE:
                cycle = dfs_visit(neighbor, current_path)
                if cycle:
```

```
                    return cycle

        colors[node] = NodeColor.BLACK
        return None

    for node in adjacency_list:
        if colors[node] == NodeColor.WHITE:
            cycle = dfs_visit(node, [])
            if cycle:
                return CycleDetectionResult(True, cycle)

    return CycleDetectionResult(False)

def topological_sort_kahns(adjacency_list: Dict[str, List[str]]) -> List[List[str]]:
    """
    Perform topological sort using Kahn's algorithm.
    Returns list of lists where each inner list contains tasks that can run in parallel.
    """
    # Calculate in-degrees
    in_degree = defaultdict(int)
    all_nodes = set(adjacency_list.keys())
    for node in adjacency_list:
        for neighbor in adjacency_list[node]:
            all_nodes.add(neighbor)
            in_degree[neighbor] += 1

    # Initialize queue with nodes having zero in-degree
    queue = deque([node for node in all_nodes if in_degree[node] == 0])
    result_levels = []

    while queue:
        # All nodes in current queue can run in parallel
        current_level = list(queue)
```

```
        result_levels.append(current_level)
        queue.clear()

        # Process current level nodes
        for node in current_level:
            for neighbor in adjacency_list.get(node, []):
                in_degree[neighbor] -= 1
                if in_degree[neighbor] == 0:
                    queue.append(neighbor)

    return result_levels

def calculate_critical_path(adjacency_list: Dict[str, List[str]],
                            task_durations: Dict[str, int]) -> Tuple[List[str], int]:
    """
    Calculate the critical path (longest path) through the DAG.
    Returns tuple of (critical_path_nodes, total_duration).
    """
    # Build reverse adjacency list for backward traversal
    reverse_adj = defaultdict(list)
    all_nodes = set(adjacency_list.keys())

    for node in adjacency_list:
        for neighbor in adjacency_list[node]:
            reverse_adj[neighbor].append(node)
            all_nodes.add(neighbor)

    # Calculate longest path to each node using dynamic programming
    longest_path = {}
    path_predecessor = {}

    def calculate_longest_path_to_node(node: str) -> int:
        if node in longest_path:
```

```
        return longest_path[node]

    if not reverse_adj[node]:  # No predecessors - starting node
        longest_path[node] = task_durations.get(node, 0)
        return longest_path[node]

    max_path = 0
    best_predecessor = None

    for predecessor in reverse_adj[node]:
        pred_path = calculate_longest_path_to_node(predecessor)
        if pred_path > max_path:
            max_path = pred_path
            best_predecessor = predecessor

    longest_path[node] = max_path + task_durations.get(node, 0)
    path_predecessor[node] = best_predecessor
    return longest_path[node]

# Find the node with maximum longest path (end of critical path)
max_duration = 0
critical_end = None

for node in all_nodes:
    duration = calculate_longest_path_to_node(node)
    if duration > max_duration:
        max_duration = duration
        critical_end = node

# Reconstruct critical path
critical_path = []
current = critical_end
while current is not None:
```

```
        critical_path.append(current)
        current = path_predecessor.get(current)

    critical_path.reverse()
    return critical_path, max_duration
```

D. Core Logic Skeleton Code:

Pipeline Parser (`dag/parser.py`):

PYTHON

```python
import yaml

import json

from typing import Dict, Any, List, Optional

from datetime import datetime

from .models import PipelineDefinition, TaskDefinition, RetryPolicy

from .exceptions import PipelineParsingError

import jsonschema


class PipelineParser:

    def __init__(self, schema_path: str):

        """Initialize parser with JSON schema for validation."""

        with open(schema_path, 'r') as f:

            self.schema = json.load(f)



    def parse_yaml_file(self, file_path: str) -> PipelineDefinition:

        """

        Parse pipeline definition from YAML file.

        Validates schema and constructs PipelineDefinition object.

        """

        # TODO 1: Load YAML file and handle file reading errors gracefully

        # TODO 2: Validate loaded YAML against JSON schema using jsonschema.validate()

        # TODO 3: Extract pipeline metadata (id, name, description, schedule, parameters)

        # TODO 4: Parse tasks list and create TaskDefinition objects for each task

        # TODO 5: Handle template parameter substitution if parameters are provided

        # TODO 6: Create and return PipelineDefinition object with all parsed data

        # TODO 7: Wrap any parsing errors in PipelineParsingError with helpful messages



        # Hint: Use yaml.safe_load() to avoid security issues with arbitrary Python execution

        # Hint: jsonschema.validate(data, schema) raises ValidationError with detailed messages

        pass



    def parse_python_definition(self, definition_func) -> PipelineDefinition:

        """
```

```
        Parse pipeline definition from Python function that returns pipeline dict.
        Supports dynamic pipeline generation with parameters.
        """
        # TODO 1: Call definition_func() to get pipeline dictionary
        # TODO 2: Handle any exceptions from dynamic pipeline generation
        # TODO 3: Validate returned dictionary against JSON schema
        # TODO 4: Convert dictionary to PipelineDefinition object (reuse logic from YAML parser)
        # TODO 5: Return constructed PipelineDefinition

        # Hint: Python definitions enable dynamic task generation based on runtime conditions
        pass

    def _create_task_definition(self, task_dict: Dict[str, Any]) -> TaskDefinition:
        """Convert task dictionary to TaskDefinition object with full validation."""
        # TODO 1: Extract required fields (id, name, type, config) with validation
        # TODO 2: Parse optional dependencies list, defaulting to empty list
        # TODO 3: Parse retry_policy dict and create RetryPolicy object, using defaults if not provided
        # TODO 4: Parse timeout_seconds with reasonable default (e.g., 3600 seconds)
        # TODO 5: Validate that task type is one of supported types (extract, transform, load, validate, notify)
        # TODO 6: Create and return TaskDefinition object

        # Hint: Use .get() method with defaults for optional fields
        # Hint: Validate task_dict keys against expected schema before accessing
        pass

    def _substitute_parameters(self, template_str: str, parameters: Dict[str, Any]) -> str:
        """
        Substitute template variables in configuration strings.
        Supports {{variable}} and {{variable|default}} syntax.
        """
        # TODO 1: Find all template variables using regex pattern {{variable_name}} or {{variable_name|default}}
        # TODO 2: For each template variable, check if parameter exists in parameters dict
        # TODO 3: If parameter exists, substitute with actual value
```

```
        # TODO 4: If parameter doesn't exist but default is provided, use default value
        # TODO 5: If parameter doesn't exist and no default, raise PipelineParsingError
        # TODO 6: Return string with all substitutions completed


        # Hint: Use re.findall() to find template patterns, then re.sub() for substitution
        pass
```

DAG Validator (`dag/validator.py`):

PYTHON

```python
from typing import List, Dict, Set
from .models import PipelineDefinition
from .graph_utils import detect_cycles_dfs, CycleDetectionResult
from .exceptions import PipelineValidationError

class DAGValidator:
    def validate_pipeline(self, pipeline: PipelineDefinition) -> List[str]:
        """
        Comprehensive pipeline validation returning list of error messages.
        Empty list indicates valid pipeline.
        """
        errors = []

        # TODO 1: Validate pipeline-level constraints (unique ID, valid schedule format)
        # TODO 2: Build task ID set and check for duplicates within pipeline
        # TODO 3: Validate each task individually (call _validate_task for each)
        # TODO 4: Build dependency graph from task definitions
        # TODO 5: Validate dependency references (all dependencies must exist as tasks)
        # TODO 6: Perform cycle detection on dependency graph
        # TODO 7: Check for orphaned tasks (tasks with no path from root tasks)
        # TODO 8: Validate resource constraints if specified
        # TODO 9: Return accumulated error list

        # Hint: Build adjacency list: {task_id: [list_of_dependent_task_ids]}
        # Hint: Use set operations to find missing dependency references efficiently
        return errors

    def _validate_task(self, task: TaskDefinition, all_task_ids: Set[str]) -> List[str]:
        """Validate individual task definition and return any errors found."""
        errors = []

        # TODO 1: Validate task ID format (alphanumeric, underscore, hyphen only)
        # TODO 2: Validate task name is non-empty
```

```
        # TODO 3: Validate task type is supported (extract, transform, load, validate, notify)
        # TODO 4: Check that dependencies list contains only valid task IDs from all_task_ids
        # TODO 5: Validate retry policy values (max_attempts > 0, backoff_seconds >= 0)
        # TODO 6: Validate timeout_seconds is reasonable (> 0, < 86400)
        # TODO 7: Validate task-specific config based on task type

        # Hint: Use regular expressions for ID format validation
        # Hint: Task-specific config validation can be extended based on task types
        return errors

    def _build_adjacency_list(self, pipeline: PipelineDefinition) -> Dict[str, List[str]]:
        """Build adjacency list representation of task dependency graph."""
        # TODO 1: Initialize adjacency list dictionary with all task IDs as keys
        # TODO 2: For each task, add its dependencies as outgoing edges
        # TODO 3: Handle tasks with no dependencies (empty adjacency list)
        # TODO 4: Return complete adjacency list for graph algorithms

        # Hint: Adjacency list maps each task to list of tasks that depend on it (downstream tasks)
        pass

    def _validate_dependency_references(self, pipeline: PipelineDefinition) -> List[str]:
        """Check that all dependency references point to existing tasks."""
        # TODO 1: Build set of all valid task IDs in pipeline
        # TODO 2: For each task, check that all its dependencies exist in task ID set
        # TODO 3: Collect any invalid dependency references
        # TODO 4: Return list of error messages for invalid references
        # TODO 5: Suggest similar task names for typos using string similarity

        # Hint: Use set difference to find invalid references efficiently
        pass

    def _detect_orphaned_tasks(self, adjacency_list: Dict[str, List[str]]) -> List[str]:
        """Find tasks that have no path from any root task (tasks with no dependencies)."""
```

```
        # TODO 1: Identify root tasks (tasks with no incoming dependencies)
        # TODO 2: Perform BFS/DFS from all root tasks to find reachable tasks
        # TODO 3: Compare reachable tasks with all tasks to find orphaned ones
        # TODO 4: Return list of orphaned task IDs


        # Hint: Build reverse adjacency list to find tasks with no incoming edges
        pass
```

Execution Planner (`dag/execution_planner.py`):

PYTHON

```python
from typing import List, Dict, Optional, Tuple
from .models import PipelineDefinition, TaskDefinition
from .graph_utils import topological_sort_kahns, calculate_critical_path

class ExecutionPlan:
    def __init__(self):
        self.execution_levels: List[List[str]] = []  # Tasks grouped by execution level
        self.critical_path: List[str] = []           # Tasks on critical path
        self.estimated_duration: int = 0             # Total estimated runtime
        self.resource_requirements: Dict[str, int] = {}  # Resource needs per level

class ExecutionPlanner:
    def __init__(self, max_parallel_tasks: int = 10):
        self.max_parallel_tasks = max_parallel_tasks

    def create_execution_plan(self, pipeline: PipelineDefinition,
                              task_durations: Optional[Dict[str, int]] = None) -> ExecutionPlan:
        """
        Create optimized execution plan with parallelization and resource awareness.
        Returns ExecutionPlan with detailed scheduling information.
        """
        # TODO 1: Build adjacency list from pipeline task dependencies
        # TODO 2: Perform topological sort to get execution levels
        # TODO 3: Apply resource constraints to limit parallelism within each level
        # TODO 4: Calculate critical path if task durations provided
        # TODO 5: Estimate total execution time based on critical path and parallelism
        # TODO 6: Generate resource requirement estimates for each execution level
        # TODO 7: Create and return ExecutionPlan object with all scheduling data

        # Hint: Use topological_sort_kahns() from graph_utils for level-based execution
        # Hint: Resource constraints may split execution levels into smaller batches
        pass

    def _apply_resource_constraints(self, execution_levels: List[List[str]],
```

```
                                pipeline: PipelineDefinition) -> List[List[str]]:
        """Split execution levels that exceed resource constraints into smaller batches."""
        # TODO 1: For each execution level, check if task count exceeds max_parallel_tasks
        # TODO 2: If level is too large, split into multiple sub-levels
        # TODO 3: Consider task resource requirements (CPU, memory) when splitting
        # TODO 4: Maintain dependency ordering when creating sub-levels
        # TODO 5: Return modified execution levels with resource constraints applied

        # Hint: Simple approach is to split levels by max_parallel_tasks
        # Advanced: Consider actual resource requirements from task configs
        pass

    def _estimate_execution_time(self, execution_levels: List[List[str]],
                                 task_durations: Dict[str, int]) -> int:
        """Calculate estimated total execution time based on critical path and parallelism."""
        # TODO 1: Sum the maximum task duration within each execution level
        # TODO 2: Add estimated overhead for task startup and coordination
        # TODO 3: Consider resource contention delays for large parallel levels
        # TODO 4: Return total estimated duration in seconds

        # Hint: Each level's duration is the maximum duration of tasks in that level
        pass
```

E. Language-Specific Hints:

- **NetworkX Integration**: Use `nx.DiGraph()` for graph representation and `nx.is_directed_acyclic_graph()` for quick cycle checking, but implement custom algorithms for better error reporting
- **YAML Parsing**: Use `yaml.safe_load()` instead of `yaml.load()` to prevent arbitrary code execution security vulnerabilities
- **JSON Schema Validation**: Install `jsonschema` package and use detailed error messages from `ValidationError.message` for user-friendly feedback
- **Template Substitution**: Use `re.findall(r'\{\{([^}]+)\}\}', template_str)` to find template variables, then `str.replace()` for substitution
- **File Path Handling**: Use `pathlib.Path` for cross-platform file operations and `Path.exists()` for file validation
- **Error Aggregation**: Collect all validation errors before returning to provide comprehensive feedback rather than failing on first error

F. Milestone Checkpoint:

After implementing DAG parsing and validation:

1. **Basic Functionality Test**:

```bash
python -m pytest pipeline_engine/dag/tests/test_parser.py -v

python -m pytest pipeline_engine/dag/tests/test_validator.py -v
```

2. **Integration Test**: Create a test pipeline YAML file and verify parsing:

```python
from pipeline_engine.dag.parser import PipelineParser

from pipeline_engine.dag.validator import DAGValidator


parser = PipelineParser('schemas/pipeline_schema.json')

validator = DAGValidator()


# Should parse successfully

pipeline = parser.parse_yaml_file('tests/fixtures/simple_pipeline.yaml')

errors = validator.validate_pipeline(pipeline)

assert len(errors) == 0, f"Validation errors: {errors}"


# Should detect cycle in invalid pipeline

invalid_pipeline = parser.parse_yaml_file('tests/fixtures/invalid_pipeline.yaml')

errors = validator.validate_pipeline(invalid_pipeline)

assert len(errors) > 0, "Should have detected validation errors"
```

3. **Expected Output**:

- Valid pipelines parse without errors and produce `PipelineDefinition` objects
- Invalid pipelines return specific error messages mentioning cycle detection, missing dependencies, or schema violations
- Visualization generates SVG or HTML files showing task nodes and dependency edges

4. **Signs of Issues**:

- **Parsing fails silently**: Check schema validation is working and exceptions are properly handled
- **Cycle detection gives false positives**: Verify adjacency list construction - dependencies should point FROM upstream TO downstream
- **Topological sort produces wrong order**: Check that in-degree calculation counts incoming dependencies correctly

Data Extraction and Loading

> **Milestone(s):** Milestone 2 - Data Extraction & Loading: Implements extractors and loaders for common data sources with incremental loading, schema mapping, and bulk transfer capabilities.

Mental Model: Universal Adapters

Think of data connectors like universal power adapters when traveling internationally. Just as a universal adapter has a standard plug interface on one side and swappable country-specific plugs on the other, our ETL connectors have a standard internal data interface and pluggable source/destination-specific implementations.

The universal adapter abstracts away the complexity of different electrical systems (voltage, frequency, plug shape) and provides a consistent interface to your device. Similarly, our data connectors abstract away the complexity of different data systems (authentication, pagination, data formats, network protocols) and provide a consistent `DataStream` interface to the transformation engine.

When you plug your laptop into a universal adapter, you don't need to know whether you're connected to 120V or 240V power - the adapter handles that complexity. When your ETL pipeline calls `extract(connection, query)`, it doesn't need to know whether it's reading from PostgreSQL, a REST API, or a CSV file - the connector handles that complexity and returns a standardized data stream.

Just as a good universal adapter is reliable (won't fry your electronics), efficient (minimal power loss), and resumable (maintains connection through brief outages), our data connectors must be reliable (handle network failures gracefully), efficient (stream data without loading everything into memory), and resumable (support incremental loading and checkpointing).

Source Connectors

Source connectors implement the extraction logic for reading data from various systems. Each connector abstracts the complexity of its specific data source while providing a uniform interface to the ETL pipeline. The connector architecture uses a plugin pattern where each source type implements a common `SourceConnector` interface.

Database Connectors handle relational databases like PostgreSQL, MySQL, and SQL Server. These connectors manage database connections, execute SQL queries, and stream results efficiently. The database connector must handle connection pooling to avoid overwhelming the source system with too many concurrent connections. For large result sets, the connector implements cursor-based pagination to avoid loading millions of rows into memory at once.

The database connector supports both full extraction (reading all data) and incremental extraction (reading only changed data since the last run). For incremental extraction, the connector uses watermarking strategies based on timestamp columns, auto-incrementing IDs, or change data capture (CDC) logs. The connector automatically detects the optimal incremental strategy by analyzing the table schema for suitable watermark columns.

API Connectors interface with REST APIs and handle the complexities of HTTP communication, authentication, rate limiting, and pagination. API responses often use different pagination patterns - some use offset/limit, others use cursor tokens, and some use page numbers. The API connector automatically detects the pagination pattern from the initial response and adapts its extraction strategy accordingly.

Authentication is handled through pluggable authentication providers that support API keys, OAuth 2.0, JWT tokens, and basic authentication. The connector includes automatic token refresh logic for OAuth flows and respects rate limits by implementing exponential backoff when receiving 429 (Too Many Requests) responses.

File System Connectors read data from various file formats including CSV, JSON, Parquet, and Avro files. These connectors can read from local file systems, network shares, or cloud storage systems like S3, GCS, or Azure Blob Storage. The file connector automatically detects file formats based on extensions and content sniffing, then selects the appropriate parser.

For large files, the connector implements streaming readers that process data in chunks rather than loading entire files into memory. This is particularly important for multi-gigabyte CSV files or large JSON arrays. The connector supports file globbing patterns to read multiple related files as a single logical dataset.

Decision: Pluggable Connector Architecture

- **Context**: Different data sources have vastly different connection patterns, authentication schemes, and data formats, making a one-size-fits-all approach impractical
- **Options Considered**: Monolithic connector with conditional logic, pluggable architecture with common interface, separate tools for each source type
- **Decision**: Pluggable architecture with abstract base class and concrete implementations
- **Rationale**: Enables independent development of connectors, easier testing and maintenance, and allows third-party connector development without modifying core system
- **Consequences**: Requires well-designed interface abstraction but provides maximum flexibility and extensibility

Connector Type	Authentication Methods	Pagination Support	Incremental Loading	Schema Detection
Database	Username/password, connection strings, SSL certificates	Cursor-based, offset/limit	Timestamp, ID-based, CDC	Information schema queries
REST API	API keys, OAuth 2.0, JWT, basic auth	Cursor tokens, offset/limit, page numbers	Modified-since headers, cursor bookmarks	Content-Type headers, response introspection
File System	Access keys, service accounts, network credentials	File chunking, directory scanning	File modification time, manifest files	File extension, content sniffing

The source connector interface defines a standard contract that all implementations must follow:

Method	Parameters	Returns	Description
`connect()`	connection_config: Dict	Connection	Establishes connection to source system with retry logic
`extract()`	query: Query, options: ExtractOptions	DataStream	Extracts data matching query parameters as streaming iterator
`get_schema()`	table_name: str	Schema	Retrieves schema information including column names, types, constraints
`supports_incremental()`	table_name: str	bool	Indicates whether source supports incremental extraction
`get_watermark()`	table_name: str	Optional[Any]	Returns current high-water mark for incremental extraction
`test_connection()`	connection_config: Dict	ConnectionTestResult	Validates connection configuration without establishing full connection

Change Data Capture (CDC) Integration represents an advanced incremental loading strategy where the source system publishes a stream of data changes (inserts, updates, deletes) rather than requiring the ETL system to poll for changes. Many modern databases support CDC through features like PostgreSQL's logical replication, MySQL's binlog, or SQL Server's Change Tracking.

CDC connectors subscribe to these change streams and convert database-specific change events into standardized change records. Each change record includes the operation type (INSERT, UPDATE, DELETE), the affected row data, and metadata like transaction IDs and timestamps. This approach provides near real-time data synchronization with minimal impact on the source system.

However, CDC comes with operational complexity. The connector must handle stream interruptions gracefully, maintain proper offset tracking to avoid missing or duplicating changes, and deal with schema evolution when source tables are altered. The CDC connector

implements checkpointing to periodically save its position in the change stream, enabling recovery from exactly where it left off after failures.

Destination Connectors

Destination connectors implement the loading logic for writing data to target systems. Like source connectors, they follow a pluggable architecture but focus on optimizing write performance, handling schema mapping, and ensuring data consistency during loads.

Database Destination Connectors write data to relational databases using bulk loading techniques optimized for each database system. PostgreSQL destinations use `COPY` commands for maximum throughput, while MySQL destinations use `LOAD DATA INFILE` or batch `INSERT` statements. The connector automatically selects the optimal loading strategy based on the destination database type and data volume.

Schema mapping is a critical function where the connector translates between source and destination data types. For example, a source API might return timestamps as ISO 8601 strings, but the destination PostgreSQL table expects `TIMESTAMP WITH TIME ZONE` columns. The connector maintains mapping rules that define how to convert between different data type systems.

Upsert operations (insert or update) require careful handling of conflict resolution. The connector must identify which columns constitute the primary key or unique constraint, then generate appropriate `ON CONFLICT` (PostgreSQL) or `ON DUPLICATE KEY UPDATE` (MySQL) statements. For databases that don't support native upsert syntax, the connector implements upsert logic using separate insert and update operations with proper transaction boundaries.

Data Warehouse Connectors are specialized for analytical systems like Snowflake, BigQuery, or Redshift. These systems optimize for different usage patterns than transactional databases - they favor bulk loading over row-by-row operations and often require specific file formats for optimal performance.

The data warehouse connector implements staging-based loading where data is first written to temporary staging tables, then merged into final destination tables using SQL `MERGE` statements. This pattern enables atomic updates where either the entire batch succeeds or fails together, preventing partial loads that could corrupt analytical queries.

Column-oriented data warehouses often perform better with specific file formats. The connector can export data to Parquet files for systems like BigQuery, or generate CSV files with appropriate delimiters and escaping for Snowflake's `COPY INTO` command. The connector handles the complexity of file generation, upload to cloud storage, and triggering the warehouse's native bulk loading commands.

Stream Processing Connectors write data to real-time systems like Apache Kafka, Amazon Kinesis, or Google Pub/Sub. These connectors must handle message ordering, partitioning strategies, and delivery guarantees. Unlike batch-oriented database connectors, stream connectors must consider message size limits, serialization formats, and consumer scaling patterns.

The stream connector implements configurable partitioning strategies to ensure related records are delivered to the same partition for ordered processing. For example, all changes for a specific customer might be routed to the same Kafka partition using a hash of the customer ID. This ensures downstream consumers can process changes in the correct order.

> **Decision: Staging-Based Loading for Data Warehouses**
>
> - **Context**: Data warehouses optimize for analytical queries and bulk operations, while row-by-row loading creates small files and poor query performance
> - **Options Considered**: Direct row-by-row inserts, streaming API calls, staging table with bulk merge operations
> - **Decision**: Staging-based loading with temporary tables and bulk merge operations
> - **Rationale**: Provides atomicity (all-or-nothing loading), optimal performance through bulk operations, and enables data validation before final commit
> - **Consequences**: Requires additional storage for staging tables but significantly improves loading performance and data consistency

Destination Type	Loading Strategy	Conflict Resolution	Transaction Support	Optimal Batch Size
PostgreSQL	COPY commands, batch INSERT	ON CONFLICT clauses	Full ACID transactions	10,000-50,000 rows
MySQL	LOAD DATA, batch INSERT	ON DUPLICATE KEY UPDATE	Full ACID transactions	5,000-25,000 rows
Snowflake	COPY INTO from staged files	MERGE statements	Warehouse-level consistency	1M+ rows per file
BigQuery	Streaming inserts, load jobs	Table decorators, DML MERGE	Eventually consistent	10MB-1GB per load job
Kafka	Producer API with batching	Idempotent producers	At-least-once delivery	100-1000 messages

The destination connector interface mirrors the source connector pattern with methods optimized for write operations:

Method	Parameters	Returns	Description
`connect()`	connection_config: Dict	Connection	Establishes connection to destination system with write permissions
`load()`	data_stream: DataStream, target: str, options: LoadOptions	LoadResult	Writes data stream to target table/topic with specified options
`create_target()`	target: str, schema: Schema	bool	Creates target table/topic if it doesn't exist
`get_target_schema()`	target: str	Schema	Retrieves current schema of target table for validation
`supports_upsert()`	target: str	bool	Indicates whether destination supports upsert operations
`begin_transaction()`	None	Transaction	Starts transaction for atomic multi-table loading
`commit_transaction()`	transaction: Transaction	bool	Commits transaction and makes changes permanent

Bulk Loading Optimization is crucial for achieving high throughput in ETL pipelines. The destination connector implements several optimization techniques based on the target system's capabilities. For databases, this includes disabling indexes during loading, using unlogged tables for temporary data, and parallelizing writes across multiple connections.

The connector monitors loading performance and dynamically adjusts batch sizes based on throughput metrics. If small batches are causing excessive overhead, the connector increases batch size up to memory limits. If large batches are causing timeouts or memory issues, the connector reduces batch size to maintain stability.

Connection pooling is essential for parallel loading scenarios where multiple tasks might write to the same destination simultaneously. The connector maintains a pool of database connections that can be shared across concurrent loading operations while ensuring proper isolation and transaction boundaries.

Incremental Loading Strategies

Incremental loading is the practice of extracting only data that has changed since the last successful pipeline run, rather than reprocessing the entire dataset. This dramatically improves pipeline performance and reduces load on source systems, but requires careful state management and change detection strategies.

Watermarking is the most common incremental loading pattern, where the pipeline tracks a "high-water mark" representing the latest processed data point. The watermark is typically based on a monotonically increasing column like a timestamp, auto-increment ID, or version number. On each pipeline run, the system extracts only records where the watermark column is greater than the stored high-water mark.

Timestamp-based watermarking uses columns like `created_at` or `modified_at` to identify new or changed records. The pipeline stores the maximum timestamp from the previous run and extracts records with timestamps greater than this value. However, timestamp watermarking has subtleties around clock skew, transaction timing, and null values that must be handled carefully.

Consider a scenario where records are inserted with `created_at = '2024-01-15 14:30:00'` but the previous pipeline run completed at `14:30:05`. If the pipeline uses `WHERE created_at > '2024-01-15 14:30:05'` for the next extraction, it will miss records that were inserted during the brief overlap period. To handle this, the watermarking strategy typically includes a small lookback window (e.g., 5 minutes) to ensure no records are missed due to timing issues.

Cursor-Based Pagination provides a more robust alternative to timestamp watermarking by using opaque cursor tokens that represent positions in the data stream. Many APIs provide cursor tokens that encode the exact position of the last returned record, allowing the next request to continue from that exact point without gaps or duplicates.

The cursor approach handles several edge cases that timestamp watermarking struggles with. If multiple records have identical timestamps, cursor-based pagination can still distinguish between them. If records are updated rather than inserted, cursors can track the update sequence. If the source system experiences clock changes or time zone transitions, cursors remain unaffected.

However, cursor-based pagination requires the source system to maintain stable cursors across API calls. Some systems generate time-limited cursors that expire after a certain period, requiring the pipeline to fall back to full extraction if too much time passes between runs. The incremental loading logic must handle cursor expiration gracefully and maintain fallback strategies.

Change Data Capture (CDC) Watermarking represents the most sophisticated incremental loading strategy, where the source system publishes a stream of change events that the pipeline can consume incrementally. Each change event contains metadata like log sequence numbers (LSNs) or transaction IDs that serve as watermarks for tracking progress through the change stream.

CDC watermarking provides several advantages over other strategies. It captures all types of changes (inserts, updates, deletes) rather than just new records. It provides near real-time latency since changes are available immediately rather than waiting for batch extraction windows. It eliminates the need to query the source system for changes since changes are pushed rather than pulled.

The complexity of CDC watermarking lies in handling stream failures and recovery. If the pipeline crashes or loses connection to the change stream, it must be able to resume from exactly where it left off without missing or duplicating changes. This requires persistent storage of watermark positions and careful handling of duplicate detection during recovery periods.

> **Decision: Multi-Strategy Incremental Loading**
>
> - **Context**: Different source systems provide different mechanisms for change detection, and no single strategy works optimally for all sources
> - **Options Considered**: Single timestamp-based strategy, single cursor-based strategy, multi-strategy approach with automatic selection
> - **Decision**: Multi-strategy approach where connectors automatically select the best available strategy
> - **Rationale**: Maximizes compatibility with diverse source systems while optimizing performance for each system's capabilities
> - **Consequences**: Increases connector complexity but provides optimal incremental loading for each source type

Strategy	Best Use Cases	Advantages	Limitations	Recovery Complexity
Timestamp Watermarking	Tables with reliable created_at/modified_at columns	Simple implementation, works with any SQL database	Clock skew issues, may miss concurrent inserts	Low - just store last timestamp
ID-Based Watermarking	Tables with auto-increment primary keys	No clock dependencies, handles concurrent inserts	Only captures new records, not updates	Low - store last processed ID
Cursor-Based Pagination	APIs with stable cursor support	No gaps or duplicates, handles complex ordering	Requires API cursor support, cursors may expire	Medium - handle cursor expiration
Change Data Capture	Systems with CDC capabilities	Real-time, captures all change types including deletes	Complex setup, requires CDC infrastructure	High - manage stream offsets and recovery

Watermark Storage and Management requires careful consideration of where and how to store watermark values. The watermark must be stored transactionally with the loaded data to ensure consistency. If the data load succeeds but the watermark update fails, the next pipeline run will reprocess the same data, potentially causing duplicates. If the watermark update succeeds but the data load fails, the next pipeline run will skip data that was never actually loaded.

The watermarking system stores watermarks in a dedicated metadata table with columns for pipeline ID, source table, watermark column, and watermark value. Each watermark entry includes timestamps for when it was created and last updated, enabling audit trails and debugging of incremental loading issues.

For complex pipelines that extract from multiple source tables, the watermarking system maintains separate watermarks for each source table. This allows different tables to have different incremental strategies and prevents failures in one table from affecting watermark management for other tables.

Backfill and Historical Loading represents a special case of incremental loading where the pipeline needs to process historical data that predates the current watermark. This might be necessary when adding new data sources, recovering from extended outages, or reprocessing data after bug fixes.

The backfill process creates temporary watermarks that start from a specified historical point and incrementally process data in chunks up to the current watermark. This allows historical processing to use the same incremental loading logic as regular pipeline runs while maintaining proper progress tracking.

Backfill operations must coordinate with regular pipeline runs to avoid conflicts. The system implements backfill scheduling that ensures historical processing doesn't interfere with current data extraction, typically by running backfill jobs during low-traffic periods or using separate resource pools.

Schema Evolution Handling becomes critical in incremental loading scenarios where the source or destination schema changes between pipeline runs. The incremental loading system must detect schema changes and handle them gracefully without breaking the watermarking logic.

When new columns are added to source tables, the incremental loading logic continues to work with existing watermark columns. However, the pipeline must handle cases where watermark columns themselves are modified or removed. The system maintains schema fingerprints alongside watermarks to detect when schema changes might affect incremental loading strategies.

For schema changes that break existing watermarks (such as changing the watermark column data type), the system provides schema migration tools that can reset watermarks and optionally trigger backfill operations to ensure data consistency.

⚠️ Pitfall: Watermark Clock Skew Issues

A common mistake is using server timestamps as watermarks without accounting for clock differences between the source system and the ETL pipeline. If the source database server's clock is 5 minutes ahead of the ETL server's clock, timestamp-based watermarking

may miss records that appear to be "in the future" from the ETL system's perspective. Always use the source system's clock for timestamp comparisons and include appropriate lookback windows to handle minor clock skew.

⚠️ Pitfall: Non-Atomic Watermark Updates

Another frequent error is updating the watermark in a separate transaction from the data loading operation. This creates a race condition where the system might crash after loading data but before updating the watermark, causing data duplication on the next run. Alternatively, updating the watermark before confirming successful data loading can cause data loss if the load fails. Always update watermarks atomically with data loading operations using database transactions or equivalent consistency mechanisms.

⚠️ Pitfall: Ignoring Null Watermark Values

Many developers forget to handle null values in watermark columns, which can cause incremental loading queries to behave unexpectedly. In SQL, comparisons with null values return unknown rather than true or false, potentially excluding records with null timestamps from incremental loads. Design incremental loading queries to explicitly handle null values in watermark columns and decide whether to include or exclude them based on business requirements.

Implementation Guidance

The data extraction and loading system requires careful balance between flexibility and performance. This section provides practical guidance for implementing connectors that can handle diverse data sources while maintaining high throughput and reliability.

Technology Recommendations:

Component	Simple Option	Advanced Option
Database Connectivity	`psycopg2` for PostgreSQL, `mysql-connector-python` for MySQL	`SQLAlchemy` with connection pooling and dialect abstraction
HTTP Client	`requests` library with session reuse	`aiohttp` for async operations or `httpx` for HTTP/2 support
File Processing	`csv` module, `json` module for basic formats	`pandas` for complex transformations, `pyarrow` for Parquet
Cloud Storage	`boto3` for AWS S3, individual cloud SDKs	`fsspec` for unified interface across cloud providers
Streaming	`itertools` for basic iteration patterns	`apache-beam` or `dask` for distributed stream processing

Recommended File Structure:

```
etl-system/
  src/connectors/
    __init__.py                    ← connector registry and factory functions
    base.py                        ← abstract base classes for source/destination connectors
    source/
      __init__.py                  ← source connector imports and registry
      database.py                  ← database source connector implementation
      api.py                       ← REST API source connector implementation
      filesystem.py                ← file system source connector implementation
    destination/
      __init__.py                  ← destination connector imports and registry
      database.py                  ← database destination connector implementation
      warehouse.py                 ← data warehouse connector implementation
      streaming.py                 ← stream processing connector implementation
    utils/
      __init__.py
      pagination.py                ← pagination strategy implementations
      watermark.py                 ← watermark management utilities
      schema.py                    ← schema mapping and validation utilities
  tests/connectors/
    test_source_database.py        ← database connector tests with test database
    test_destination_warehouse.py ← warehouse connector tests with mock services
    integration/                   ← integration tests with real services
  config/
    connector-examples/            ← example connector configurations
```

Infrastructure Starter Code:

Complete watermark management utility that handles persistent storage and atomic updates:

PYTHON

```python
from datetime import datetime, timezone
from typing import Optional, Any, Dict
from dataclasses import dataclass
import sqlite3
import json

@dataclass
class WatermarkEntry:
    pipeline_id: str
    source_table: str
    watermark_column: str
    watermark_value: Any
    watermark_type: str
    created_at: datetime
    updated_at: datetime

class WatermarkManager:
    """Manages watermark persistence and atomic updates for incremental loading."""

    def __init__(self, db_path: str = "watermarks.db"):
        self.db_path = db_path
        self._init_database()

    def _init_database(self):
        """Initialize watermark storage database with proper schema."""
        with sqlite3.connect(self.db_path) as conn:
            conn.execute("""
                CREATE TABLE IF NOT EXISTS watermarks (
                    pipeline_id TEXT NOT NULL,
                    source_table TEXT NOT NULL,
                    watermark_column TEXT NOT NULL,
                    watermark_value TEXT NOT NULL,
                    watermark_type TEXT NOT NULL,
                    created_at TEXT NOT NULL,
```

```
                updated_at TEXT NOT NULL,
                PRIMARY KEY (pipeline_id, source_table, watermark_column)
            )
        """)
        conn.execute("""
            CREATE INDEX IF NOT EXISTS idx_pipeline_table
            ON watermarks(pipeline_id, source_table)
        """)

    def get_watermark(self, pipeline_id: str, source_table: str,
                      watermark_column: str) -> Optional[Any]:
        """Retrieve current watermark value for specified source."""
        with sqlite3.connect(self.db_path) as conn:
            cursor = conn.execute(
                "SELECT watermark_value, watermark_type FROM watermarks "
                "WHERE pipeline_id = ? AND source_table = ? AND watermark_column = ?",
                (pipeline_id, source_table, watermark_column)
            )
            row = cursor.fetchone()
            if row:
                value_str, value_type = row
                return self._deserialize_watermark(value_str, value_type)
            return None

    def update_watermark(self, pipeline_id: str, source_table: str,
                         watermark_column: str, watermark_value: Any) -> bool:
        """Atomically update watermark value."""
        now = datetime.now(timezone.utc).isoformat()
        value_str = self._serialize_watermark(watermark_value)
        value_type = type(watermark_value).__name__

        with sqlite3.connect(self.db_path) as conn:
            conn.execute("""
```

```
            INSERT OR REPLACE INTO watermarks
            (pipeline_id, source_table, watermark_column, watermark_value,
             watermark_type, created_at, updated_at)
            VALUES (?, ?, ?, ?, ?,
                    COALESCE((SELECT created_at FROM watermarks
                              WHERE pipeline_id = ? AND source_table = ?
                              AND watermark_column = ?), ?), ?)
        """, (pipeline_id, source_table, watermark_column, value_str, value_type,
              pipeline_id, source_table, watermark_column, now, now))
        return True


def _serialize_watermark(self, value: Any) -> str:
    """Convert watermark value to storable string format."""
    if isinstance(value, datetime):
        return value.isoformat()
    elif isinstance(value, (int, float, str)):
        return str(value)
    else:
        return json.dumps(value)


def _deserialize_watermark(self, value_str: str, value_type: str) -> Any:
    """Convert stored string back to original watermark type."""
    if value_type == "datetime":
        return datetime.fromisoformat(value_str)
    elif value_type == "int":
        return int(value_str)
    elif value_type == "float":
        return float(value_str)
    elif value_type == "str":
        return value_str
    else:
        return json.loads(value_str)
```

Complete connection pooling utility for database connectors:

PYTHON

```python
import threading

from contextlib import contextmanager

from typing import Dict, Any, Generator

from queue import Queue, Empty

import psycopg2

from psycopg2 import pool


class ConnectionPoolManager:

    """Thread-safe connection pool manager for database connectors."""


    def __init__(self, connection_config: Dict[str, Any],

                 min_connections: int = 1, max_connections: int = 10):

        self.connection_config = connection_config

        self.min_connections = min_connections

        self.max_connections = max_connections

        self._pool = None

        self._lock = threading.Lock()

        self._initialize_pool()


    def _initialize_pool(self):

        """Create database connection pool with specified parameters."""

        try:

            self._pool = psycopg2.pool.ThreadedConnectionPool(

                minconn=self.min_connections,

                maxconn=self.max_connections,

                host=self.connection_config['host'],

                port=self.connection_config.get('port', 5432),

                database=self.connection_config['database'],

                user=self.connection_config['user'],

                password=self.connection_config['password']

            )

        except Exception as e:

            raise ConnectionError(f"Failed to initialize connection pool: {e}")
```

```
    @contextmanager
    def get_connection(self) -> Generator[Any, None, None]:
        """Get connection from pool with automatic cleanup."""
        conn = None
        try:
            with self._lock:
                conn = self._pool.getconn()
            yield conn
        except Exception as e:
            if conn:
                conn.rollback()
            raise e
        finally:
            if conn:
                with self._lock:
                    self._pool.putconn(conn)


    def close_all_connections(self):
        """Close all connections in the pool."""
        if self._pool:
            self._pool.closeall()
```

Core Logic Skeleton Code:

Database source connector implementation with detailed TODOs for learners:

PYTHON

```python
from typing import Iterator, Dict, Any, Optional
from abc import ABC, abstractmethod
import pandas as pd

class SourceConnector(ABC):
    """Abstract base class for all source connectors."""

    @abstractmethod
    def extract(self, query: str, options: Dict[str, Any]) -> Iterator[Dict[str, Any]]:
        """Extract data from source system as iterator of records."""
        pass

    @abstractmethod
    def get_schema(self, table_name: str) -> Dict[str, str]:
        """Get schema information for specified table."""
        pass

class DatabaseSourceConnector(SourceConnector):
    """Source connector for relational databases with incremental loading support."""

    def __init__(self, connection_config: Dict[str, Any]):
        self.connection_config = connection_config
        self.pool_manager = None
        self.watermark_manager = WatermarkManager()

    def extract(self, query: str, options: Dict[str, Any]) -> Iterator[Dict[str, Any]]:
        """Extract data from database with optional incremental loading."""
        # TODO 1: Initialize connection pool if not already created
        # TODO 2: Check if incremental loading is requested in options
        # TODO 3: If incremental, modify query to include watermark WHERE clause
        # TODO 4: Execute query using connection from pool
        # TODO 5: Fetch results in batches to avoid memory issues (use fetchmany)
        # TODO 6: Yield each row as dictionary with column names as keys
        # TODO 7: Track maximum watermark value seen during extraction
```

```
        # TODO 8: After successful extraction, update stored watermark
        # Hint: Use pandas.read_sql with chunksize parameter for large result sets
        # Hint: Convert each row tuple to dict using column names from cursor.description
        pass


    def get_schema(self, table_name: str) -> Dict[str, str]:
        """Retrieve column information from database system tables."""
        # TODO 1: Connect to database using connection pool
        # TODO 2: Query information_schema.columns for table structure
        # TODO 3: Handle different database types (PostgreSQL, MySQL, etc.)
        # TODO 4: Map database-specific types to standard type names
        # TODO 5: Return dictionary mapping column names to standardized types
        # Hint: PostgreSQL uses information_schema.columns, MySQL uses DESCRIBE
        # Hint: Common type mapping: VARCHAR->str, INTEGER->int, TIMESTAMP->datetime
        pass


    def _build_incremental_query(self, base_query: str, watermark_info: Dict[str, Any]) -> str:
        """Modify base query to include watermark filtering for incremental extraction."""
        # TODO 1: Parse base query to identify WHERE clause location
        # TODO 2: Add watermark condition (e.g., modified_at > last_watermark)
        # TODO 3: Handle case where base query already has WHERE clause
        # TODO 4: Include small lookback window to handle clock skew
        # TODO 5: Return modified query string
        # Hint: Use SQL parsing library or simple string manipulation
        # Hint: Always use parameterized queries to prevent SQL injection
        pass
```

API source connector with pagination and rate limiting:

PYTHON

```python
import requests

import time

from typing import Iterator, Dict, Any, Optional

from requests.adapters import HTTPAdapter

from urllib3.util.retry import Retry


class APISourceConnector(SourceConnector):

    """Source connector for REST APIs with automatic pagination and rate limiting."""



    def __init__(self, connection_config: Dict[str, Any]):

        self.base_url = connection_config['base_url']

        self.auth_config = connection_config.get('auth', {})

        self.session = self._create_session()



    def extract(self, endpoint: str, options: Dict[str, Any]) -> Iterator[Dict[str, Any]]:

        """Extract data from API endpoint with automatic pagination handling."""

        # TODO 1: Authenticate with API using configured auth method

        # TODO 2: Make initial request to endpoint with base parameters

        # TODO 3: Detect pagination pattern from response (cursor, offset, page-based)

        # TODO 4: Extract records from response data based on configured data_path

        # TODO 5: Yield individual records from current page

        # TODO 6: Check for next page indicator and prepare next request

        # TODO 7: Implement rate limiting with exponential backoff on 429 errors

        # TODO 8: Continue pagination until no more pages available

        # Hint: Common pagination patterns - look for 'next', 'cursor', 'has_more' fields

        # Hint: Use time.sleep() between requests to respect rate limits

        pass



    def _create_session(self) -> requests.Session:

        """Create HTTP session with retry strategy and timeout configuration."""

        # TODO 1: Create requests.Session object

        # TODO 2: Configure retry strategy for transient failures

        # TODO 3: Set appropriate timeouts for connect and read operations
```

```
        # TODO 4: Add authentication headers/parameters based on auth_config
        # TODO 5: Mount HTTPAdapter with retry configuration
        # Hint: Retry on 500, 502, 503, 504 status codes but not 4xx errors
        # Hint: Use exponential backoff starting from 1 second
        pass

    def _handle_pagination(self, response: Dict[str, Any]) -> Optional[Dict[str, Any]]:
        """Determine next page parameters based on response pagination metadata."""
        # TODO 1: Check response for pagination indicators (next_cursor, next_page, etc.)
        # TODO 2: Extract pagination parameters for next request
        # TODO 3: Return None if no more pages available
        # TODO 4: Handle different pagination patterns (offset/limit, cursor-based, page numbers)
        # TODO 5: Validate pagination parameters to avoid infinite loops
        # Hint: Store pagination state to detect when you've seen the same cursor twice
        pass
```

Milestone Checkpoint:

After implementing the source and destination connectors, verify functionality with these tests:

1. **Database Connector Test**: Create a test PostgreSQL database with sample data. Run `python -m pytest tests/connectors/test_source_database.py` and verify:
   - Full extraction returns all records
   - Incremental extraction with timestamp watermark returns only new records
   - Schema detection correctly identifies column types
   - Connection pooling handles concurrent extractions
2. **API Connector Test**: Use a public API like JSONPlaceholder (https://jsonplaceholder.typicode.com/). Run extraction and verify:
   - Pagination automatically follows all pages
   - Rate limiting respects API limits without errors
   - Authentication headers are properly included
   - Cursor-based pagination maintains position across requests
3. **Integration Test**: Set up end-to-end pipeline from database source to database destination:

```bash
python scripts/test_pipeline.py --source postgres://test_db/users --dest postgres://target_db/users_copy
```

Expected output shows successful extraction, transformation, and loading with record counts.

Common Debugging Issues:

Symptom	Likely Cause	Diagnosis	Fix
Incremental extraction misses records	Clock skew between systems	Check timestamps on both source and ETL servers	Add 5-10 minute lookback window to watermark queries
API extraction fails with 429 errors	Rate limiting too aggressive	Check API rate limit headers in response	Implement exponential backoff with longer delays
Database connection timeout	Connection pool exhausted	Monitor active connection count	Increase pool size or reduce query timeout
Partial data loads in destination	Non-atomic watermark updates	Check if watermark is updated before load completion	Use database transactions to update watermark with data
Memory errors during large extractions	Loading entire result set into memory	Check if using streaming vs batch loading	Implement cursor-based iteration with smaller batch sizes

Data Transformation Engine

> **Milestone(s):** Milestone 3 - Data Transformations: Implements transformation operations with schema validation, supporting SQL-based transformations, Python UDFs, null handling, and data validation rules.

Mental Model: Data Refinery

Think of the data transformation engine as an **oil refinery** that processes raw crude oil into various refined products. Just as a refinery has multiple processing units (distillation towers, crackers, reformers) that each perform specific transformations on the petroleum, our data transformation engine has multiple transformation stages that clean, reshape, and enrich raw data.

In an oil refinery, crude oil enters the facility and flows through a series of specialized processing units. Each unit has specific operating parameters (temperature, pressure, catalysts) and produces intermediate products that feed into downstream units. Quality control labs continuously test the products at each stage, rejecting batches that don't meet specifications and sending them back for reprocessing.

Similarly, our data transformation engine receives raw extracted data and passes it through a pipeline of transformation stages. Each stage has configurable parameters (SQL queries, Python functions, validation rules) and produces intermediate datasets. Schema validation acts like quality control, checking data types, null constraints, and business rules at each stage, flagging or rejecting records that don't meet specifications.

The key insight from this analogy is that **transformations are composable and order-dependent**. Just as you can't crack heavy oil before distilling it, you often need to clean and standardize data before applying complex business logic transformations. The refinery analogy also highlights that **quality control must be continuous** - checking only the final output misses problems that compound through multiple stages.

This mental model guides our design decisions: transformation stages should be pluggable and reusable, intermediate results should be inspectable for debugging, and validation should happen at stage boundaries to catch problems early rather than at the end of a long transformation pipeline.

SQL-Based Transformations

SQL-based transformations form the backbone of most ETL operations because SQL provides a declarative, widely-understood language for data manipulation. The transformation engine treats SQL as a **templating language** where queries can contain parameter substitutions, enabling reusable transformation logic across different pipeline runs and datasets.

Decision: Templated SQL with Runtime Parameter Substitution

- **Context**: ETL pipelines need to apply similar transformations across different time periods, datasets, or environments, requiring dynamic query generation
- **Options Considered**: Static SQL files, string concatenation, template engine (Jinja2), prepared statement parameters
- **Decision**: Jinja2 template engine for SQL with runtime parameter substitution
- **Rationale**: Template engines provide safe parameter substitution preventing SQL injection, support complex logic (loops, conditionals), and are familiar to data engineers. Prepared statements only handle value parameters, not structural changes like table names or column lists.
- **Consequences**: Enables parameterized queries but requires template validation and introduces dependency on Jinja2. Complex templates can become hard to debug.

Template Feature	Purpose	Example Usage	Validation Required
Variable Substitution	Dynamic table/column names	`SELECT * FROM {{ source_table }}`	Table existence check
Date Range Filters	Incremental processing	`WHERE created_at >= '{{ start_date }}'`	Date format validation
Conditional Logic	Environment-specific queries	`{% if env == 'prod' %} AND status = 'active' {% endif %}`	Branch coverage testing
Loop Constructs	Dynamic column generation	`{% for col in numeric_cols %} SUM({{ col }}) {% endfor %}`	Column existence validation
Macro Functions	Reusable query fragments	`{{ standardize_phone_number('phone_col') }}`	Macro parameter validation

The SQL transformation executor follows a multi-phase execution model that separates template rendering from SQL execution, enabling better error handling and debugging:

1. **Template Validation Phase**: Parse the SQL template using Jinja2 parser to check for syntax errors, undefined variables, and template structure issues before runtime
2. **Parameter Injection Phase**: Merge runtime parameters from pipeline configuration, upstream task outputs, and system variables into template context
3. **Template Rendering Phase**: Generate the final SQL query by applying parameters to template, producing executable SQL with all variables resolved
4. **Query Validation Phase**: Validate the rendered SQL for syntax correctness, check referenced tables/columns exist, and verify the user has required permissions
5. **Execution Phase**: Execute the SQL against the target database, handling connection pooling, transaction management, and result set streaming
6. **Result Processing Phase**: Convert database result sets into standardized data structures, apply any post-processing transformations, and pass results to downstream tasks

> The critical insight here is that template rendering and SQL execution are separate phases - this allows us to generate and inspect the final SQL before execution, enabling better debugging and dry-run capabilities.

Parameterization Strategy

The transformation engine supports multiple parameter sources with a defined precedence order to handle conflicts:

Parameter Source	Precedence	Example Usage	When to Use
Task Configuration	1 (highest)	Task-specific table names, custom business logic	Task-specific overrides
Pipeline Parameters	2	Environment settings, global date ranges	Pipeline-wide configuration
System Variables	3	Current timestamp, pipeline run ID, execution date	Built-in system context
Environment Variables	4 (lowest)	Database connection strings, API keys	Infrastructure configuration

The parameter resolution engine merges these sources into a single context dictionary, with higher precedence sources overriding lower ones. This enables flexible configuration inheritance where global settings provide defaults that specific tasks can override as needed.

Query Result Handling

SQL transformations produce tabular results that must be converted into the standard `DataStream` format for pipeline interoperability. The transformation engine handles this conversion while managing memory efficiently for large result sets:

Result Processing Strategy	Memory Usage	Throughput	When to Use
Streaming Row-by-Row	Low	Medium	Large result sets, memory constraints
Batched Processing	Medium	High	Moderate result sets, balanced performance
Full Materialization	High	Highest	Small result sets, need random access

The executor automatically chooses the appropriate strategy based on result set size estimates from query plan analysis. For result sets under 10MB, it uses full materialization for maximum performance. For larger sets, it switches to batched processing with configurable batch sizes, typically 1000-10000 rows per batch depending on row width.

Error Handling and Rollback

SQL transformations run within database transactions to ensure consistency and enable rollback on failures:

1. **Connection Acquisition**: Obtain database connection from connection pool, handling pool exhaustion with exponential backoff retry
2. **Transaction Begin**: Start explicit transaction to ensure atomicity of all SQL operations within the transformation
3. **Query Execution**: Execute the transformed SQL, capturing both result data and execution metadata (row counts, execution time)
4. **Validation Checks**: Verify result data meets expected schema and business rule constraints before committing
5. **Transaction Commit**: Commit the transaction if all validations pass, making changes permanent
6. **Connection Release**: Return connection to pool for reuse, ensuring proper cleanup regardless of success or failure

If any step fails, the transaction automatically rolls back, leaving the database in its original state. This transactional approach is crucial for data consistency, especially when transformations modify multiple tables or when downstream tasks depend on complete, consistent datasets.

Common Pitfalls in SQL Transformations

⚠️ **Pitfall: Template Injection Vulnerabilities** Using string concatenation or unsafe templating can create SQL injection vulnerabilities. For example, `SELECT * FROM users WHERE name = '{{ user_input }}'` allows malicious input to break out of the string context. Always use Jinja2's auto-escaping features and validate parameter values against expected patterns before template rendering.

⚠️ **Pitfall: Resource Exhaustion from Large Result Sets** Materializing large query results in memory can exhaust available RAM and crash the pipeline executor. Monitor query result size estimates and automatically switch to streaming processing for results over configurable thresholds. Implement query result limits as a safety mechanism.

⚠️ **Pitfall: Transaction Timeout in Long-Running Queries** Database transactions held open for extended periods can block other operations and may be automatically rolled back by database timeout settings. For transformations that take more than a few minutes, consider breaking them into smaller chunks or using separate transactions for independent operations.

Python User-Defined Functions

Python User-Defined Functions (UDFs) enable custom transformation logic that goes beyond SQL capabilities, such as complex string processing, machine learning inference, external API calls, or specialized business logic. The UDF execution engine provides a safe, performant environment for running Python code within the transformation pipeline while maintaining data type safety and error isolation.

> **Decision: Isolated Python Execution with Resource Limits**
>
> - **Context**: Python UDFs need to execute arbitrary user code safely without compromising pipeline stability or security
> - **Options Considered**: Same-process execution, subprocess isolation, Docker containers, serverless functions
> - **Decision**: Subprocess isolation with resource limits and timeout enforcement
> - **Rationale**: Subprocess isolation prevents memory leaks and crashes from affecting the main pipeline process, while resource limits prevent runaway UDFs from consuming excessive CPU/memory. Simpler than containers but provides adequate isolation.
> - **Consequences**: Enables safe execution of arbitrary Python code but adds process overhead and inter-process communication complexity. Subprocess startup time affects performance for small datasets.

UDF Definition and Registration

Python UDFs are defined as regular Python functions with type annotations that specify input and output schemas. The transformation engine uses these annotations for automatic data type conversion and validation:

UDF Component	Purpose	Example	Validation Applied
Function Signature	Define input parameters and types	`def clean_phone(phone: str) -> str:`	Parameter count and types
Type Annotations	Specify expected data types	`phone: Optional[str]` for nullable columns	Null handling requirements
Docstring Schema	Document expected behavior	`"""Standardizes phone numbers to E.164 format"""`	Human-readable documentation
Return Type	Specify output data type	`-> Optional[str]`	Output type validation
Error Handling	Manage exceptions	`try/except` blocks with specific error types	Exception classification

The UDF registry maintains a catalog of available functions with their signatures, enabling the pipeline definition parser to validate UDF usage at pipeline definition time rather than runtime. This early validation prevents many common errors like type mismatches or missing function definitions.

Data Type Mapping and Conversion

The transformation engine automatically converts between the pipeline's internal data representation and Python native types, handling null values, type coercion, and precision requirements:

Pipeline Type	Python Type	Null Handling	Precision Notes
`String`	`str`	`None` for null	Unicode encoding preserved
`Integer`	`int`	`None` for null	Unlimited precision in Python
`Float`	`float`	`None` for null, `NaN` for invalid	IEEE 754 double precision
`Decimal`	`decimal.Decimal`	`None` for null	Arbitrary precision maintained
`Boolean`	`bool`	`None` for null	Strict true/false, no truthy conversion
`Date`	`datetime.date`	`None` for null	ISO date format
`Timestamp`	`datetime.datetime`	`None` for null	Timezone-aware UTC
`JSON`	`dict` or `list`	`None` for null	Nested structures supported

The type conversion engine handles edge cases like integer overflow, floating-point precision loss, and timezone conversion automatically. For cases where automatic conversion might lose information (like converting `Decimal` to `float`), it logs warnings and provides configuration options to make the conversion explicit or reject the operation.

UDF Execution Modes

The transformation engine supports different execution modes for Python UDFs based on performance requirements and data characteristics:

Execution Mode	Use Case	Performance Characteristics	Memory Usage
Row-by-Row	Complex logic per record	Low throughput, high latency	Constant
Vectorized	Pandas/NumPy operations	High throughput, moderate latency	Proportional to batch size
Streaming	Large datasets	Moderate throughput, low latency	Constant
Batch	Bulk operations	Highest throughput, high latency	High

Row-by-Row Mode executes the UDF once per input record, suitable for complex transformations that require full context of each record. This mode has the lowest performance but provides the most flexibility and is easiest to debug.

Vectorized Mode passes entire columns as pandas Series or NumPy arrays to the UDF, enabling efficient operations on numerical data. UDFs in this mode must be written to handle array inputs and produce array outputs with the same length.

Streaming Mode processes data in small batches (typically 100-1000 records) while maintaining constant memory usage. This balances performance with resource consumption for medium-sized datasets.

Batch Mode materializes the entire input dataset before passing it to the UDF, enabling operations that require full dataset context like statistical analysis or machine learning training.

Resource Management and Safety

Python UDF execution includes comprehensive resource management to prevent runaway processes and ensure predictable performance:

1. **Memory Limits**: Each UDF subprocess has a configurable memory limit (default 1GB) enforced through OS process limits, preventing individual functions from exhausting system memory
2. **CPU Limits**: CPU time limits prevent infinite loops or computationally intensive operations from blocking pipeline progress indefinitely
3. **Timeout Enforcement**: Wall-clock timeout ensures UDFs complete within reasonable time bounds, even if waiting on I/O operations

4. **Subprocess Isolation**: Each UDF runs in a separate Python subprocess, preventing crashes or memory leaks from affecting the main pipeline process
5. **Import Restrictions**: UDF execution environments restrict imports to approved libraries, preventing access to dangerous modules like `os`, `subprocess`, or `sys`
6. **Network Isolation**: By default, UDF processes cannot make outbound network connections, though this can be configured for specific use cases

The resource management system monitors UDF execution continuously and terminates processes that exceed limits, logging detailed information about resource usage for performance tuning.

Error Handling and Debugging

UDF error handling follows a structured approach that maximizes debugging information while maintaining pipeline reliability:

Error Type	Handling Strategy	Information Captured	Recovery Action
Import Error	Fail fast at registration	Missing module, Python version	Pipeline validation failure
Type Error	Record-level handling	Expected vs actual types, record ID	Skip record or fail task
Value Error	Configurable handling	Invalid input value, validation rules	Apply default or fail record
Runtime Exception	Isolation and logging	Full stack trace, input data	Fail record, continue batch
Resource Exhaustion	Process termination	Resource usage stats, partial results	Retry with smaller batch
Timeout	Graceful shutdown	Execution time, processed records	Retry or skip based on config

The error handling system provides multiple recovery strategies:

- **Fail Fast**: Stop pipeline execution immediately on first error, suitable for critical transformations where partial results are unacceptable
- **Skip Invalid Records**: Continue processing valid records while logging invalid ones for later analysis
- **Apply Default Values**: Replace failed transformations with configured default values, useful for optional enrichment operations
- **Retry with Backoff**: Retry failed UDF calls with exponential backoff for transient errors like temporary resource constraints

Common Pitfalls in Python UDFs

⚠️ **Pitfall: Memory Leaks from Global Variables** Python UDFs that use global variables or class-level state can accumulate memory across multiple invocations since subprocess environments are reused. Always use function-local variables and explicitly clean up any resources like file handles or database connections within the UDF.

⚠️ **Pitfall: Inconsistent Null Handling** Different Python libraries handle `None` values differently - pandas uses `NaN`, NumPy has multiple null representations, and standard library functions may raise exceptions. Always explicitly check for `None` in UDF inputs and decide how to handle null values consistently with your data model.

⚠️ **Pitfall: Side Effects in Pure Transformation Functions** UDFs should be pure functions that don't modify external state, write files, or make network calls unless explicitly designed for those purposes. Side effects make UDFs non-idempotent and can cause inconsistent results when pipelines retry or run in parallel.

Schema Validation and Evolution

Schema validation ensures data consistency throughout the transformation pipeline by checking data types, constraints, and business rules at transformation boundaries. The validation engine must balance thoroughness with performance while handling the inevitable evolution of data schemas over time.

Schema evolution presents one of the most challenging aspects of ETL system design because it requires maintaining backward compatibility while allowing data models to grow and change. The transformation engine treats schema as a **first-class citizen** with

explicit versioning, migration strategies, and compatibility checking.

> **Decision: Schema Registry with Backward Compatibility Checking**
>
> - **Context**: Data schemas evolve over time as business requirements change, requiring systematic management of schema versions and compatibility
> - **Options Considered**: No schema management, inline schema definitions, external schema registry, database-driven schema evolution
> - **Decision**: Centralized schema registry with version management and compatibility checking
> - **Rationale**: Centralized registry enables schema reuse across pipelines, version management tracks evolution over time, and compatibility checking prevents breaking changes from propagating. External systems like Confluent Schema Registry provide proven solutions.
> - **Consequences**: Enables systematic schema evolution but adds operational complexity and external dependency. Schema registry becomes critical infrastructure component.

Schema Definition and Versioning

The schema registry maintains versioned schema definitions that specify data structure, constraints, and evolution rules:

Schema Component	Purpose	Example	Validation Applied
Field Definitions	Column names, types, nullability	`{"name": "user_id", "type": "integer", "nullable": false}`	Type compatibility, null constraints
Primary Keys	Unique identification	`{"primary_key": ["user_id", "timestamp"]}`	Uniqueness validation
Foreign Keys	Referential integrity	`{"foreign_key": {"field": "user_id", "references": "users.id"}}`	Reference validation
Check Constraints	Business rules	`{"check": "age >= 0 AND age <= 150"}`	Value range validation
Default Values	Missing data handling	`{"field": "status", "default": "active"}`	Type-compatible defaults
Evolution Rules	Schema change policies	`{"allow_new_fields": true, "allow_field_deletion": false}`	Compatibility enforcement

Each schema version includes a compatibility level that determines what changes are allowed:

- **Full Compatibility**: New schema can read data written with any previous schema version, and previous versions can read data written with new schema
- **Backward Compatibility**: New schema can read data written with previous schema versions, but not vice versa
- **Forward Compatibility**: Previous schema versions can read data written with new schema, but new schema may not read old data
- **No Compatibility**: Breaking changes allowed, requiring explicit data migration

Validation Pipeline Architecture

The validation engine operates as a series of validation stages that data passes through during transformation:

1. **Input Validation Stage**: Validate data entering transformation against source schema, checking basic type compatibility and required field presence
2. **Constraint Validation Stage**: Apply business rule constraints like range checks, format validation, and cross-field dependencies
3. **Type Coercion Stage**: Attempt automatic type conversions for compatible types, logging all coercions for audit purposes
4. **Output Validation Stage**: Validate transformation results against target schema before passing to next pipeline stage

5. **Schema Evolution Check**: Compare current data schema against registry to detect schema drift and compatibility issues

Each stage can be configured with different error handling policies:

Validation Policy	Behavior	Use Cases	Performance Impact
Strict Validation	Fail pipeline on first validation error	Critical financial data, regulatory compliance	Lowest throughput
Best Effort	Log errors but continue processing valid records	Data exploration, non-critical analytics	Balanced performance
Schema Inference	Automatically adapt schema based on observed data	Prototype pipelines, exploratory analysis	Highest throughput
Sampling Validation	Validate random sample of records	Large datasets with consistent structure	Minimal impact

Type System and Coercion Rules

The transformation engine uses a rich type system that captures both logical data types and physical representation requirements:

Logical Type	Physical Types	Coercion Rules	Precision Handling
Numeric	`int32`, `int64`, `float32`, `float64`, `decimal`	Widening conversions only	Maintain maximum precision
Text	`varchar(n)`, `text`, `char(n)`	Truncate with warning	Preserve UTF-8 encoding
Temporal	`date`, `timestamp`, `timestamptz`	Parse standard formats	Maintain timezone info
Boolean	`bool`, `bit`, `tinyint`	Standard truthy conversion	Map to true/false
JSON	`json`, `jsonb`, `text`	Parse and validate JSON syntax	Preserve nested structure
Binary	`blob`, `bytea`, `varbinary`	Base64 encoding for text transport	Preserve exact bytes

Automatic type coercion follows safe conversion rules that never lose information without explicit user configuration. For example, `int32` to `int64` conversion is automatic, but `int64` to `int32` requires explicit truncation handling since it may lose data.

Null Value Semantics

Different data systems have varying concepts of null values, requiring careful handling during transformations:

System	Null Representation	Semantics	Transformation Behavior
SQL Databases	`NULL`	Unknown value, not equal to anything including itself	Preserve SQL null semantics
JSON	`null` literal	Explicit null value	Map to SQL NULL
CSV Files	Empty string or missing field	Ambiguous - could be null or empty	Configurable interpretation
Python	`None` object	Absence of value	Direct mapping to SQL NULL
Pandas	`NaN` (Not a Number)	Missing numerical value	Convert to SQL NULL
Apache Parquet	Null bitmap	Efficient null encoding	Direct null preservation

The transformation engine provides configurable null handling policies to manage these semantic differences:

- **Strict Null Preservation**: Maintain exact null semantics from source system
- **Null Unification**: Convert all null representations to standard SQL NULL
- **Null Replacement**: Replace nulls with configurable default values based on data type
- **Null Rejection**: Treat null values as validation errors for non-nullable fields

Schema Evolution Strategies

The schema evolution engine supports multiple strategies for handling schema changes over time:

Evolution Strategy	Approach	Benefits	Limitations
Append-Only	Only add new optional fields	Maintains full compatibility	Cannot remove obsolete fields
Versioned Schemas	Maintain multiple schema versions simultaneously	Supports breaking changes	Complexity increases with versions
Schema Migration	Transform data during schema updates	Clean schema evolution	Requires migration downtime
Schema Union	Merge schemas using union types	Handles diverse data sources	Complex type system

Append-Only Evolution is the simplest approach where new fields can be added but existing fields cannot be removed or have their types changed. This maintains backward compatibility but can lead to schema bloat over time.

Versioned Schemas allow breaking changes by maintaining multiple schema versions and routing data through appropriate transformation pipelines. Each pipeline version handles a specific schema version, enabling gradual migration.

Schema Migration performs bulk transformation of existing data when schemas change incompatibly. This requires coordinated downtime but results in clean, consistent schemas.

Schema Union approaches treat schemas as unions of possible field sets, enabling pipelines to handle multiple schema versions simultaneously. This works well for diverse data sources but requires complex type resolution logic.

Performance Optimization for Validation

Schema validation can become a performance bottleneck for high-throughput pipelines, requiring optimization strategies:

1. **Validation Caching**: Cache validation results for identical records to avoid repeated validation work
2. **Parallel Validation**: Distribute validation across multiple threads or processes for CPU-bound operations
3. **Lazy Validation**: Defer expensive validation operations until data is actually accessed downstream
4. **Sampling Strategies**: Validate statistical samples rather than every record for consistent data sources
5. **Early Exit Optimization**: Stop validation on first error for fail-fast policies
6. **Columnar Validation**: Validate entire columns at once using vectorized operations when possible

The validation engine monitors its own performance and can automatically adjust validation strategies based on observed data patterns and performance requirements.

Common Pitfalls in Schema Validation

⚠️ **Pitfall: Over-Strict Validation Causing Brittleness** Extremely strict validation rules make pipelines brittle to minor schema changes that don't affect downstream consumers. For example, rejecting records because a rarely-used optional field changes type, even though most consumers ignore that field. Design validation rules to focus on fields actually used by downstream systems.

⚠️ **Pitfall: Validation Performance Degradation** Validating every field of every record can become extremely expensive for wide tables with complex constraints. Profile validation performance and use sampling or lazy validation for non-critical checks. Monitor validation overhead as a percentage of total pipeline runtime.

⚠️ **Pitfall: Inconsistent Error Handling Across Schema Changes** Different types of schema validation failures may require different handling strategies, but implementing inconsistent policies makes debugging difficult. Establish clear error handling hierarchies: syntax errors always fail, type errors configurable, constraint violations logged but allowed to pass.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
SQL Templating	Python string formatting	Jinja2 template engine
Python UDF Execution	Same-process function calls	Subprocess isolation with resource limits
Schema Registry	JSON files in version control	Confluent Schema Registry or custom REST API
Type Validation	Manual isinstance() checks	Pydantic models with automatic validation
Database Connectivity	Direct database drivers	SQLAlchemy with connection pooling

Recommended File Structure

```
etl-pipeline/
  src/transformation/
    __init__.py
    engine.py                    ← Main transformation engine
    sql_transformer.py           ← SQL template processing
    python_udf.py               ← Python UDF execution
    schema_validator.py         ← Schema validation logic
    type_system.py              ← Type definitions and coercion
    tests/
      test_sql_transformer.py   ← SQL transformation tests
      test_python_udf.py        ← UDF execution tests
      test_schema_validator.py  ← Schema validation tests
  schemas/
    registry/
      user_events_v1.json       ← Schema definitions
      user_events_v2.json
    migrations/
      v1_to_v2_migration.sql    ← Schema evolution scripts
  transformations/
    sql/
      user_cleanup.sql          ← SQL transformation templates
      daily_aggregation.sql
    udfs/
      phone_standardization.py  ← Python UDF definitions
      address_geocoding.py
```

Infrastructure Starter Code

Here's a complete type system foundation for the transformation engine:

PYTHON

```python
# type_system.py - Complete type system implementation

from typing import Any, Dict, List, Optional, Union, Type

from enum import Enum

from datetime import datetime, date

from decimal import Decimal

import json


class DataType(Enum):

    """Supported data types in the transformation pipeline."""

    STRING = "string"

    INTEGER = "integer"

    FLOAT = "float"

    DECIMAL = "decimal"

    BOOLEAN = "boolean"

    DATE = "date"

    TIMESTAMP = "timestamp"

    JSON = "json"

    BINARY = "binary"


class ValidationResult:

    """Result of data validation operation."""

    def __init__(self, is_valid: bool, errors: List[str] = None, warnings: List[str] = None):

        self.is_valid = is_valid

        self.errors = errors or []

        self.warnings = warnings or []



    def add_error(self, error: str):

        self.errors.append(error)

        self.is_valid = False



    def add_warning(self, warning: str):

        self.warnings.append(warning)


class TypeConverter:

```

```
    """Handles type conversion between different representations."""


    def __init__(self):
        self.conversion_rules = {
            (DataType.INTEGER, DataType.FLOAT): self._int_to_float,
            (DataType.INTEGER, DataType.STRING): self._int_to_string,
            (DataType.FLOAT, DataType.STRING): self._float_to_string,
            (DataType.STRING, DataType.INTEGER): self._string_to_int,
            (DataType.STRING, DataType.FLOAT): self._string_to_float,
            (DataType.STRING, DataType.BOOLEAN): self._string_to_bool,
            (DataType.STRING, DataType.DATE): self._string_to_date,
            (DataType.STRING, DataType.TIMESTAMP): self._string_to_timestamp,
        }


    def convert(self, value: Any, from_type: DataType, to_type: DataType) -> tuple[Any, ValidationResult]:
        """Convert value from one type to another."""
        if value is None:
            return None, ValidationResult(True)


        if from_type == to_type:
            return value, ValidationResult(True)


        conversion_key = (from_type, to_type)
        if conversion_key not in self.conversion_rules:
            result = ValidationResult(False)
            result.add_error(f"No conversion rule from {from_type.value} to {to_type.value}")
            return value, result


        try:
            converted_value = self.conversion_rules[conversion_key](value)
            return converted_value, ValidationResult(True)
        except Exception as e:
            result = ValidationResult(False)
```

```
            result.add_error(f"Conversion failed: {str(e)}")
            return value, result

    def _int_to_float(self, value: int) -> float:
        return float(value)

    def _int_to_string(self, value: int) -> str:
        return str(value)

    def _float_to_string(self, value: float) -> str:
        return str(value)

    def _string_to_int(self, value: str) -> int:
        return int(value.strip())

    def _string_to_float(self, value: str) -> float:
        return float(value.strip())

    def _string_to_bool(self, value: str) -> bool:
        value_lower = value.strip().lower()
        if value_lower in ('true', '1', 'yes', 'on'):
            return True
        elif value_lower in ('false', '0', 'no', 'off'):
            return False
        else:
            raise ValueError(f"Cannot convert '{value}' to boolean")

    def _string_to_date(self, value: str) -> date:
        # Support common date formats
        for fmt in ('%Y-%m-%d', '%m/%d/%Y', '%d/%m/%Y'):
            try:
                return datetime.strptime(value.strip(), fmt).date()
            except ValueError:
```

```
                continue
        raise ValueError(f"Cannot parse date from '{value}'")

    def _string_to_timestamp(self, value: str) -> datetime:
        # Support common timestamp formats
        for fmt in ('%Y-%m-%d %H:%M:%S', '%Y-%m-%dT%H:%M:%S', '%Y-%m-%d %H:%M:%S.%f'):
            try:
                return datetime.strptime(value.strip(), fmt)
            except ValueError:
                continue
        raise ValueError(f"Cannot parse timestamp from '{value}'")

# Schema management utilities
class FieldDefinition:
    """Definition of a single field in a schema."""
    def __init__(self, name: str, data_type: DataType, nullable: bool = True,
                 default_value: Any = None, constraints: List[str] = None):
        self.name = name
        self.data_type = data_type
        self.nullable = nullable
        self.default_value = default_value
        self.constraints = constraints or []

class SchemaDefinition:
    """Complete schema definition for a dataset."""
    def __init__(self, name: str, version: int, fields: List[FieldDefinition]):
        self.name = name
        self.version = version
        self.fields = {field.name: field for field in fields}

    def get_field(self, name: str) -> Optional[FieldDefinition]:
        return self.fields.get(name)

    def validate_record(self, record: Dict[str, Any]) -> ValidationResult:
```

```
        """Validate a single record against this schema."""
        result = ValidationResult(True)


        # Check for required fields
        for field_name, field_def in self.fields.items():
            if not field_def.nullable and field_name not in record:
                result.add_error(f"Required field '{field_name}' is missing")
            elif field_name not in record:
                record[field_name] = field_def.default_value


        # Check for unexpected fields
        for field_name in record:
            if field_name not in self.fields:
                result.add_warning(f"Unexpected field '{field_name}' found")


        return result
```

Core Logic Skeleton Code

PYTHON

```python
# sql_transformer.py - SQL transformation engine skeleton

from typing import Dict, Any, Iterator, Optional

from jinja2 import Environment, FileSystemLoader, TemplateSyntaxError

import logging


class SQLTransformer:

    """Executes SQL-based transformations with parameter templating."""



    def __init__(self, template_dir: str, connection_manager):

        self.template_env = Environment(

            loader=FileSystemLoader(template_dir),

            autoescape=False  # SQL doesn't need HTML escaping

        )

        self.connection_manager = connection_manager

        self.logger = logging.getLogger(__name__)



    def execute_transformation(self, template_name: str, parameters: Dict[str, Any],

                               connection_name: str) -> Iterator[Dict[str, Any]]:

        """Execute a SQL transformation using a template with parameters."""

        # TODO 1: Load and validate the SQL template

        #   - Use self.template_env.get_template(template_name)

        #   - Handle TemplateNotFound exception

        #   - Return empty iterator on template errors



        # TODO 2: Render the template with parameters

        #   - Call template.render(**parameters)

        #   - Handle TemplateSyntaxError and UndefinedError

        #   - Log the rendered SQL for debugging



        # TODO 3: Acquire database connection

        #   - Use self.connection_manager.get_connection(connection_name)

        #   - Handle connection failures with appropriate retries

        #   - Ensure connection is returned to pool on completion
```

```
        # TODO 4: Execute the rendered SQL query
        #   - Begin transaction for consistency
        #   - Execute query and get cursor
        #   - Handle SQL syntax errors and database exceptions

        # TODO 5: Stream results back to caller
        #   - Fetch results in batches to manage memory
        #   - Convert database rows to dictionaries
        #   - Yield each record to maintain streaming behavior
        #   - Commit transaction on success, rollback on failure

        # Hint: Use try/finally to ensure connection cleanup
        # Hint: Log execution metrics (row count, duration)
        pass

    def validate_template(self, template_name: str, parameters: Dict[str, Any]) -> ValidationResult:
        """Validate template syntax and parameter completeness."""
        # TODO 1: Attempt to load template
        # TODO 2: Try rendering with provided parameters
        # TODO 3: Check for undefined variables
        # TODO 4: Return validation result with specific errors
        pass

# python_udf.py - Python UDF execution engine skeleton
import subprocess
import pickle
import json
from typing import Callable, List, Any, Dict
from concurrent.futures import ProcessPoolExecutor
import resource

class PythonUDFExecutor:
    """Executes Python User-Defined Functions with isolation and resource limits."""
```

```python
def __init__(self, max_workers: int = 4, memory_limit_mb: int = 1024):
    self.max_workers = max_workers
    self.memory_limit_mb = memory_limit_mb
    self.function_registry: Dict[str, Callable] = {}

def register_function(self, name: str, func: Callable):
    """Register a UDF for execution."""
    # TODO 1: Validate function signature has type annotations
    # TODO 2: Store function in registry with metadata
    # TODO 3: Validate function doesn't use restricted imports
    pass

def execute_function(self, function_name: str, data: List[Dict[str, Any]],
                     execution_mode: str = "row") -> Iterator[Dict[str, Any]]:
    """Execute a registered UDF on input data."""
    # TODO 1: Look up function in registry
    #   - Validate function exists
    #   - Get function metadata and signature

    # TODO 2: Choose execution strategy based on mode
    #   - "row": execute once per record
    #   - "batch": execute on entire dataset
    #   - "stream": execute on chunks

    # TODO 3: Prepare subprocess execution environment
    #   - Set memory limits using resource module
    #   - Prepare data serialization (pickle or json)
    #   - Set up timeout and CPU limits

    # TODO 4: Execute UDF in isolated subprocess
    #   - Use ProcessPoolExecutor for isolation
    #   - Handle subprocess timeouts and errors
```

```
        #   - Capture both results and any error messages

        # TODO 5: Process and validate results
        #   - Deserialize results from subprocess
        #   - Validate output types match function signature
        #   - Yield results maintaining input order

        # Hint: Use pickle for complex Python objects, JSON for simple data
        # Hint: Monitor subprocess resource usage and terminate if exceeded
        pass

    def _execute_in_subprocess(self, func_code: str, data: Any, limits: Dict[str, Any]) -> Any:
        """Execute function code in isolated subprocess with resource limits."""
        # TODO 1: Set resource limits (memory, CPU time)
        # TODO 2: Execute function code safely
        # TODO 3: Return results or error information
        pass

# schema_validator.py - Schema validation engine skeleton
class SchemaValidator:
    """Validates data against schema definitions with evolution support."""

    def __init__(self, schema_registry):
        self.schema_registry = schema_registry
        self.type_converter = TypeConverter()

    def validate_data_stream(self, data_stream: Iterator[Dict[str, Any]],
                             schema_name: str, schema_version: Optional[int] = None) -> Iterator[Dict[str, Any]]:
        """Validate and optionally transform data stream against schema."""
        # TODO 1: Resolve schema from registry
        #   - Get schema by name and version (latest if version is None)
        #   - Handle schema not found errors
        #   - Log schema information for debugging
```

```
        # TODO 2: Set up validation statistics tracking
        #   - Count total records processed
        #   - Count validation errors and warnings
        #   - Track performance metrics

        # TODO 3: Process each record in the stream
        #   - Validate record structure against schema
        #   - Apply type conversions where needed
        #   - Handle validation errors based on policy

        # TODO 4: Apply business rule validation
        #   - Check constraint rules (range checks, format validation)
        #   - Validate cross-field dependencies
        #   - Apply custom validation functions

        # TODO 5: Yield validated/transformed records
        #   - Apply any schema transformations
        #   - Include validation metadata if requested
        #   - Log summary statistics periodically

        # Hint: Use yield to maintain streaming behavior
        # Hint: Batch validation operations for better performance
        pass

    def check_schema_compatibility(self, old_schema: SchemaDefinition,
                                   new_schema: SchemaDefinition) -> ValidationResult:
        """Check compatibility between two schema versions."""
        # TODO 1: Compare field definitions between schemas
        # TODO 2: Check for breaking changes (removed fields, type changes)
        # TODO 3: Identify safe changes (new optional fields)
        # TODO 4: Return compatibility result with specific issues
```

```
        pass
```

Language-Specific Hints

- **Template Security**: Use Jinja2's `select_autoescape()` with `autoescape=False` for SQL templates to avoid escaping SQL syntax, but validate all parameters before rendering
- **Subprocess Communication**: Use `pickle` for serializing complex Python objects to subprocesses, but fallback to JSON for simple data types that need to be debuggable
- **Resource Limits**: Use the `resource` module to set `RLIMIT_AS` (memory) and `RLIMIT_CPU` (CPU time) limits in subprocess before executing UDF code
- **Connection Pooling**: Use SQLAlchemy's `pool_pre_ping=True` to validate connections before use, preventing stale connection errors
- **Type Validation**: Use `typing.get_type_hints()` to extract type annotations from UDF functions for automatic validation

Milestone Checkpoint

After implementing the data transformation engine, verify functionality with these steps:

1. **SQL Transformation Test**: Create a simple template `SELECT * FROM {{ table_name }} WHERE date >= '{{ start_date }}'` and verify it renders correctly with parameters
2. **UDF Registration Test**: Register a simple function like `def upper_case(text: str) -> str: return text.upper()` and verify it appears in the registry
3. **Schema Validation Test**: Define a simple schema with required and optional fields, then validate both compliant and non-compliant records
4. **Integration Test**: Run a complete transformation pipeline that extracts data, applies SQL transformations, executes a Python UDF, and validates against output schema

Expected behavior:

- Templates should render without errors and produce valid SQL
- UDFs should execute in isolation and return expected results
- Schema validation should catch type mismatches and constraint violations
- The complete pipeline should process sample data end-to-end

Signs something is wrong:

- **Template rendering fails**: Check Jinja2 template syntax and parameter names
- **UDF execution hangs**: Verify subprocess limits are set correctly and functions don't have infinite loops
- **Schema validation is slow**: Profile validation logic and consider sampling or lazy validation
- **Memory usage grows**: Check for memory leaks in UDF processes or connection pooling

Pipeline Orchestration and Monitoring

> **Milestone(s):** Milestone 4 - Pipeline Orchestration & Monitoring: Implements pipeline execution with monitoring, alerting, and lineage tracking

Mental Model: Air Traffic Control

Think of pipeline orchestration like an air traffic control (ATC) system at a busy airport. Just as ATC coordinates hundreds of flights with complex schedules, dependencies, and safety requirements, our orchestration system manages hundreds of data pipeline tasks with their own schedules, dependencies, and error handling needs.

In this analogy, each **pipeline** is like a flight route with multiple legs (tasks). The **scheduler** acts as the control tower, deciding when each flight can take off based on weather conditions (resource availability), runway capacity (system load), and air traffic patterns (other running pipelines). The **task execution engine** is like the ground crew and pilots executing each flight leg, reporting status back to the control tower.

Just as ATC must handle flight delays, cancellations, and emergency landings gracefully while keeping passengers informed, our orchestration system must handle task failures, retries, and cascading dependencies while providing clear visibility into what's happening. The **monitoring and alerting system** functions like the airport's information displays and announcement system, keeping stakeholders informed of delays, gate changes, and arrivals.

The critical insight from this analogy is that orchestration is fundamentally about **coordinating resources, managing dependencies, and maintaining visibility** across a complex distributed system. Like ATC, our system must be reliable, observable, and capable of graceful degradation when things go wrong.

Scheduler Integration

The scheduler integration component serves as the entry point for pipeline execution, responsible for determining when pipelines should run based on time-based schedules or external events. This component bridges the gap between pipeline definitions and actual execution, translating abstract scheduling requirements into concrete execution triggers.

Cron-Based Scheduling

Cron-based scheduling provides time-triggered pipeline execution using familiar cron expression syntax. The scheduler maintains an internal registry of active pipelines and their schedules, continuously evaluating upcoming execution windows.

The core scheduling algorithm operates on a polling model with configurable intervals (typically 30-60 seconds). During each poll cycle, the scheduler evaluates all registered pipelines to determine if any are due for execution. This evaluation considers the pipeline's cron expression, timezone settings, and any configured execution windows or blackout periods.

> **Decision: Polling vs Event-Driven Scheduling**
>
> - **Context**: Need to support both cron-based and event-driven pipeline triggers while maintaining simple operational semantics
> - **Options Considered**: Pure event-driven system with cron events, hybrid polling for cron with events for external triggers, pure polling for all trigger types
> - **Decision**: Hybrid approach with polling for cron-based schedules and event queues for external triggers
> - **Rationale**: Polling provides predictable resource usage and simple failure recovery for time-based schedules, while event queues enable low-latency response to external triggers without the complexity of distributed cron
> - **Consequences**: Slight delay (up to polling interval) for cron-triggered pipelines, but simplified operational model and better handling of system restarts

The scheduler maintains pipeline execution state to prevent duplicate runs and handle overlapping schedules. Each pipeline definition includes execution policies that control behavior when previous runs are still active or when schedules overlap.

Execution Policy	Behavior	Use Case
`ALLOW_CONCURRENT`	Start new run regardless of existing runs	Independent data processing
`SKIP_ON_RUNNING`	Skip execution if previous run still active	Long-running ETL processes
`CANCEL_PREVIOUS`	Terminate previous run and start new one	Real-time data updates
`QUEUE_SEQUENTIAL`	Queue execution to start after current completes	Dependent processing chains

The scheduler implements **catchup behavior** for pipelines that miss scheduled executions due to system downtime. When the scheduler restarts, it evaluates missed execution windows and can optionally trigger historical runs to maintain data consistency.

Event-Driven Triggers

Event-driven pipeline triggers enable reactive execution based on external system events such as file arrivals, database changes, or API notifications. The scheduler maintains event subscription queues for different trigger types, processing incoming events asynchronously from the cron-based scheduling loop.

Event triggers support **payload-based parameterization**, allowing external events to provide runtime parameters for pipeline execution. For example, a file arrival event can specify the file path as a pipeline parameter, enabling the same pipeline definition to process different files dynamically.

The event processing system implements **deduplication and idempotency** to handle duplicate events gracefully. Each event includes a unique identifier and optional deduplication window, preventing redundant pipeline executions for repeated notifications.

Event Source Type	Configuration	Deduplication Strategy
`FILE_ARRIVAL`	Directory path, file pattern, polling interval	File path + modification time
`DATABASE_CHANGE`	Connection, table, CDC log position	Transaction ID + sequence number
`API_WEBHOOK`	Endpoint URL, authentication, payload schema	Event ID from payload
`MESSAGE_QUEUE`	Queue name, consumer group, message format	Message ID + partition offset

Schedule Management Interface

The scheduler exposes a management interface for registering, updating, and monitoring pipeline schedules. This interface supports dynamic schedule updates without requiring system restarts, enabling operational flexibility for changing business requirements.

Method Name	Parameters	Returns	Description
`register_schedule`	`pipeline_id: str, schedule_config: ScheduleConfig`	`bool`	Register new pipeline schedule
`update_schedule`	`pipeline_id: str, schedule_config: ScheduleConfig`	`bool`	Modify existing pipeline schedule
`pause_schedule`	`pipeline_id: str, reason: str`	`bool`	Temporarily disable pipeline execution
`resume_schedule`	`pipeline_id: str`	`bool`	Re-enable paused pipeline schedule
`get_next_run_time`	`pipeline_id: str`	`Optional[datetime]`	Calculate next scheduled execution
`get_schedule_status`	`pipeline_id: str`	`ScheduleStatus`	Retrieve current schedule state

The scheduler maintains **schedule metadata** including creation timestamps, modification history, and execution statistics to support operational monitoring and debugging.

Task Execution Engine

The task execution engine orchestrates the actual execution of pipeline tasks, managing parallel execution, resource allocation, and state transitions. This component translates the abstract DAG structure into concrete task executions while maintaining dependency relationships and handling failures gracefully.

Parallel Execution Management

The execution engine implements **level-based parallelization**, executing all tasks at the same DAG level simultaneously while respecting resource constraints. This approach maximizes throughput while ensuring dependency correctness.

The engine maintains an **execution queue** for each parallelization level, populated by the topological sort results from the DAG validation engine. As tasks complete successfully, the engine evaluates downstream tasks to determine when their dependencies are satisfied and they can be queued for execution.

> **Decision: Thread Pool vs Process Pool for Task Execution**
>
> - **Context**: Tasks may include CPU-intensive transformations, I/O-bound data loading, and potentially untrusted user-defined functions
> - **Options Considered**: Single-threaded sequential execution, thread pool with shared memory, process pool with isolation, hybrid approach based on task type
> - **Decision**: Configurable hybrid approach with thread pool for I/O-bound tasks and process pool for CPU-intensive or untrusted tasks
> - **Rationale**: Thread pool provides efficient resource sharing for database connections and network I/O, while process pool ensures isolation for UDFs and prevents memory leaks from affecting other tasks
> - **Consequences**: Requires task type classification in pipeline definitions, but provides optimal performance and safety characteristics

The execution engine implements **resource-aware scheduling** to prevent system overload. Each task definition includes resource requirements (CPU cores, memory, network connections), and the engine tracks available system resources to determine execution capacity.

Resource Type	Measurement Unit	Default Limit	Overflow Behavior
`CPU_CORES`	Number of cores	System CPU count	Queue task until resources available
`MEMORY_MB`	Megabytes	75% of system memory	Queue task with memory pressure warning
`DB_CONNECTIONS`	Connection count	Connection pool size	Queue task until connection available
`NETWORK_BANDWIDTH`	Mbps	Unlimited (monitoring only)	Log warning but continue execution

State Management and Persistence

The execution engine maintains comprehensive state information for all active task executions, persisting state changes to enable recovery after system failures. The state model supports complex execution patterns including retries, cancellations, and dependency failures.

Task execution state follows a well-defined state machine with specific transition rules. The engine validates all state transitions to ensure consistency and prevent invalid operations.

Current State	Valid Events	Next State	Actions Taken
`PENDING`	`DEPENDENCIES_MET`	`QUEUED`	Add to execution queue
`QUEUED`	`EXECUTION_STARTED`	`RUNNING`	Allocate resources, start task
`RUNNING`	`EXECUTION_COMPLETED`	`SUCCESS`	Release resources, update metrics
`RUNNING`	`EXECUTION_FAILED`	`FAILED` or `RETRYING`	Check retry policy, possibly reschedule
`FAILED`	`RETRY_SCHEDULED`	`QUEUED`	Increment attempt count, apply backoff
`RETRYING`	`EXECUTION_STARTED`	`RUNNING`	Start retry attempt
`SUCCESS`	`UPSTREAM_FAILED`	`SKIPPED`	Mark downstream tasks as skipped

The engine implements **optimistic concurrency control** for state updates, using version numbers to detect concurrent modifications. This ensures state consistency when multiple components (scheduler, execution threads, monitoring) update task state simultaneously.

Resource Allocation and Cleanup

The execution engine manages system resources including database connections, temporary files, and memory allocations. Resource management follows a **lease-based model** where each task execution receives a time-bounded lease on required resources.

Database connections are managed through a **connection pooling strategy** that maintains separate pools for different database types and connection configurations. The engine pre-allocates connections based on upcoming task requirements and implements connection health checking to ensure reliability.

Connection Pool	Max Size	Idle Timeout	Health Check Interval
`PRIMARY_DB`	20 connections	300 seconds	60 seconds
`WAREHOUSE_DB`	10 connections	600 seconds	120 seconds
`STAGING_DB`	15 connections	180 seconds	30 seconds

Temporary file management implements **automatic cleanup** with configurable retention policies. The engine creates isolated temporary directories for each task execution and schedules cleanup based on task completion status and retention requirements.

⚠️ **Pitfall: Resource Leak in Failed Tasks** Many implementations fail to properly clean up resources when tasks fail unexpectedly. This leads to connection pool exhaustion, temporary disk space consumption, and memory leaks. Always implement resource cleanup in finally blocks or using context managers, and include explicit resource leak detection in monitoring dashboards. Track resource allocation per task execution and alert when cleanup doesn't occur within expected timeframes.

Monitoring and Alerting

The monitoring and alerting system provides comprehensive observability into pipeline execution, collecting metrics, logs, and lineage information to support operational management and debugging. This system operates as a separate subsystem that observes execution without interfering with performance.

Metrics Collection and Aggregation

The monitoring system collects **multi-dimensional metrics** across different aspects of pipeline execution including performance, reliability, and resource utilization. Metrics collection uses an asynchronous publishing model to minimize impact on task execution performance.

The system implements **hierarchical metric aggregation**, collecting detailed per-task metrics while rolling up summary statistics at the pipeline and system levels. This approach enables both detailed debugging and high-level operational dashboards.

Metric Category	Examples	Collection Frequency	Retention Period
`PERFORMANCE`	Task duration, queue wait time, throughput	Per task execution	90 days detailed, 1 year aggregated
`RELIABILITY`	Success rate, failure count, retry attempts	Per task execution	180 days detailed, 2 years aggregated
`RESOURCE`	CPU usage, memory consumption, I/O wait	Every 30 seconds during execution	30 days detailed, 6 months aggregated
`BUSINESS`	Records processed, data quality scores	Per task execution	1 year detailed, 5 years aggregated

The metrics system supports **custom business metrics** defined in pipeline configurations, enabling domain-specific monitoring. Tasks can emit custom metrics using a simple API that automatically handles aggregation and persistence.

Real-Time Dashboard Integration

The monitoring system provides **real-time dashboard integration** through standardized APIs that support popular monitoring platforms. The dashboard data model emphasizes actionable information over raw metrics, presenting derived insights that guide operational decisions.

The system implements **adaptive alerting thresholds** that automatically adjust based on historical performance patterns and seasonal variations. This reduces false positive alerts while maintaining sensitivity to genuine issues.

Dashboard View	Update Frequency	Key Metrics	Drill-Down Capability
`SYSTEM_OVERVIEW`	30 seconds	Active pipelines, success rate, resource utilization	Pipeline-level details
`PIPELINE_DETAIL`	15 seconds	Task status, execution timeline, dependency graph	Individual task logs
`RESOURCE_MONITORING`	10 seconds	CPU, memory, network, storage I/O	Historical trending
`DATA_QUALITY`	Per pipeline run	Validation failures, schema changes, row counts	Quality rule details

Failure Notification and Escalation

The alerting system implements **intelligent failure classification** that categorizes failures by severity, impact, and required response. This classification drives different notification channels and escalation timelines.

The system maintains **alert routing rules** that direct notifications to appropriate teams based on failure characteristics, time of day, and team availability. Integration with on-call scheduling systems ensures critical failures reach available personnel quickly.

> **Decision: Push vs Pull Alerting Model**
>
> - **Context**: Need to balance timely notifications with alert fatigue while supporting multiple notification channels
> - **Options Considered**: Pure push model with immediate notifications, pure pull model with polling-based alerts, hybrid model with intelligent routing
> - **Decision**: Hybrid model with immediate push for critical failures and batched notifications for non-critical issues
> - **Rationale**: Critical failures (data corruption, security issues) require immediate attention, while transient failures (temporary network issues) benefit from aggregation to reduce noise
> - **Consequences**: Requires failure severity classification but significantly improves alert signal-to-noise ratio and reduces operational fatigue

The notification system supports **alert suppression and correlation** to prevent alert storms during widespread system issues. When multiple related failures occur simultaneously, the system identifies root cause relationships and suppresses downstream alerts.

Alert Severity	Notification Channel	Response Time SLA	Escalation Timeline
`CRITICAL`	Phone, SMS, Slack	Immediate	15 minutes to secondary on-call
`HIGH`	Email, Slack	5 minutes	1 hour to team lead
`MEDIUM`	Email	15 minutes	4 hours to product owner
`LOW`	Dashboard only	Best effort	Weekly summary report

Data Lineage and Audit Trail

The monitoring system captures comprehensive **data lineage information** that tracks data flow through transformation steps and across pipeline boundaries. This lineage information supports impact analysis, debugging, and compliance requirements.

Lineage tracking operates at the **dataset and column level**, recording not just which tables were read and written, but which specific columns influenced which outputs. This granular tracking enables precise impact analysis when upstream data changes or issues occur.

The system maintains an **immutable audit trail** of all pipeline executions, configuration changes, and manual interventions. This audit trail supports compliance requirements and provides a complete historical record for debugging complex issues.

Lineage Event Type	Captured Information	Retention Policy
`DATA_READ`	Source table, query, row count, timestamp	2 years
`DATA_WRITE`	Target table, operation type, row count, schema	5 years
`TRANSFORMATION`	Function name, input columns, output columns, parameters	2 years
`SCHEMA_CHANGE`	Old schema, new schema, compatibility check results	10 years

⚠️ **Pitfall: Lineage Collection Performance Impact** Detailed lineage collection can significantly impact pipeline performance if not implemented carefully. Avoid synchronous lineage writes during task execution - instead, buffer lineage events in memory and flush asynchronously. Use sampling for high-throughput pipelines and implement circuit breakers to disable lineage collection if it begins affecting SLA compliance. Monitor lineage collection overhead and tune based on business requirements.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Task Queue	Redis with `rq` library	Apache Airflow or Celery with RabbitMQ
Metrics Storage	SQLite with custom tables	InfluxDB or Prometheus
Log Aggregation	File-based logging with logrotate	ELK stack (Elasticsearch, Logstash, Kibana)
Alerting	Email via `smtplib`	PagerDuty API integration
Dashboard	Flask web UI with simple HTML/CSS	Grafana with custom dashboards
State Persistence	SQLite database	PostgreSQL with connection pooling

Recommended File Structure

```
pipeline_orchestration/
├── __init__.py
├── scheduler/
│   ├── __init__.py
│   ├── cron_scheduler.py      # Time-based scheduling logic
│   ├── event_scheduler.py     # Event-driven trigger handling
│   └── schedule_manager.py    # Schedule registration and management
├── executor/
│   ├── __init__.py
│   ├── task_executor.py       # Core task execution engine
│   ├── resource_manager.py    # Resource allocation and cleanup
│   └── state_manager.py       # Task state persistence and transitions
├── monitoring/
│   ├── __init__.py
│   ├── metrics_collector.py   # Metrics collection and aggregation
│   ├── alerting.py           # Alert routing and notification
│   └── lineage_tracker.py    # Data lineage capture and storage
└── web/
    ├── __init__.py
    ├── dashboard.py          # Web dashboard for monitoring
    └── api.py               # REST API for external integrations
```

Infrastructure Starter Code

Basic Metrics Collection System:

PYTHON

```python
import time

import threading

from collections import defaultdict, deque

from datetime import datetime, timedelta

from typing import Dict, List, Optional, Any

import json

import sqlite3


class MetricsCollector:

    """Thread-safe metrics collection with automatic aggregation."""



    def __init__(self, db_path: str = "metrics.db"):

        self.db_path = db_path

        self.metrics_buffer = defaultdict(deque)

        self.lock = threading.RLock()

        self._init_db()

        self._start_background_flush()



    def _init_db(self):

        """Initialize metrics database schema."""

        with sqlite3.connect(self.db_path) as conn:

            conn.execute("""

                CREATE TABLE IF NOT EXISTS task_metrics (

                    id INTEGER PRIMARY KEY AUTOINCREMENT,

                    task_id TEXT NOT NULL,

                    pipeline_run_id TEXT NOT NULL,

                    metric_name TEXT NOT NULL,

                    metric_value REAL NOT NULL,

                    labels TEXT,  -- JSON string

                    timestamp DATETIME NOT NULL

                )

            """)

            conn.execute("""
```

```
            CREATE INDEX IF NOT EXISTS idx_metrics_task_time
            ON task_metrics(task_id, timestamp)
        """)

    def record_metric(self, task_id: str, pipeline_run_id: str,
                      metric_name: str, value: float,
                      labels: Optional[Dict[str, str]] = None):
        """Record a metric value for a specific task."""
        with self.lock:
            metric_record = {
                'task_id': task_id,
                'pipeline_run_id': pipeline_run_id,
                'metric_name': metric_name,
                'value': value,
                'labels': json.dumps(labels or {}),
                'timestamp': datetime.utcnow()
            }
            self.metrics_buffer[task_id].append(metric_record)

    def _flush_metrics(self):
        """Flush buffered metrics to database."""
        if not self.metrics_buffer:
            return

        with self.lock:
            all_metrics = []
            for task_metrics in self.metrics_buffer.values():
                all_metrics.extend(task_metrics)
            self.metrics_buffer.clear()

        if all_metrics:
            with sqlite3.connect(self.db_path) as conn:
                conn.executemany("""
```

```
                    INSERT INTO task_metrics
                    (task_id, pipeline_run_id, metric_name, metric_value, labels, timestamp)
                    VALUES (?, ?, ?, ?, ?, ?)
                """, [(m['task_id'], m['pipeline_run_id'], m['metric_name'],
                       m['value'], m['labels'], m['timestamp']) for m in all_metrics])

class SimpleStateManager:
    """Thread-safe task state management with persistence."""

    def __init__(self, db_path: str = "task_state.db"):
        self.db_path = db_path
        self.lock = threading.RLock()
        self._init_db()

    def _init_db(self):
        """Initialize state database schema."""
        with sqlite3.connect(self.db_path) as conn:
            conn.execute("""
                CREATE TABLE IF NOT EXISTS task_executions (
                    task_id TEXT NOT NULL,
                    pipeline_run_id TEXT NOT NULL,
                    state TEXT NOT NULL,
                    attempt_count INTEGER NOT NULL,
                    started_at DATETIME,
                    completed_at DATETIME,
                    error_message TEXT,
                    version INTEGER NOT NULL DEFAULT 1,
                    PRIMARY KEY (task_id, pipeline_run_id)
                )
            """)

    def create_execution(self, task_id: str, pipeline_run_id: str) -> TaskExecution:
        """Create new task execution in PENDING state."""
```

```
        execution = TaskExecution(
            task_id=task_id,
            pipeline_run_id=pipeline_run_id,
            state=TaskState.PENDING,
            attempt_count=0,
            started_at=None,
            completed_at=None,
            error_message=None,
            logs=[],
            metrics={}
        )

        with sqlite3.connect(self.db_path) as conn:
            conn.execute("""
                INSERT OR REPLACE INTO task_executions
                (task_id, pipeline_run_id, state, attempt_count)
                VALUES (?, ?, ?, ?)
            """, (task_id, pipeline_run_id, execution.state.value, execution.attempt_count))

        return execution

    def transition_state(self, task_id: str, pipeline_run_id: str,
                         event: TaskEvent, error_message: str = None) -> bool:
        """Attempt state transition based on event."""
        # Implementation details in skeleton below
        pass

class BasicAlertManager:
    """Simple alerting system with email and webhook support."""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.alert_history = deque(maxlen=1000)  # Keep recent alerts for suppression
```

```
def send_alert(self, severity: str, title: str, message: str,
               task_id: str = None, pipeline_id: str = None):
    """Send alert through configured channels based on severity."""
    alert = {
        'severity': severity,
        'title': title,
        'message': message,
        'task_id': task_id,
        'pipeline_id': pipeline_id,
        'timestamp': datetime.utcnow()
    }

    # Check for suppression (simplified)
    if self._should_suppress_alert(alert):
        return

    self.alert_history.append(alert)

    # Route based on severity
    channels = self.config.get('alert_channels', {}).get(severity, ['email'])
    for channel in channels:
        self._send_to_channel(channel, alert)
```

Core Logic Skeleton

Task Execution Engine Core:

PYTHON

```python
class TaskExecutionEngine:
    """Orchestrates parallel execution of pipeline tasks with resource management."""

    def __init__(self, max_workers: int = 4, resource_limits: Dict[str, int] = None):
        self.max_workers = max_workers
        self.resource_limits = resource_limits or {}
        self.active_executions: Dict[str, TaskExecution] = {}
        self.execution_queue = queue.Queue()
        self.metrics = MetricsCollector()
        self.state_manager = SimpleStateManager()

    def execute_pipeline(self, pipeline: PipelineDefinition,
                         run_id: str, parameters: Dict[str, Any] = None) -> bool:
        """Execute complete pipeline with dependency management and monitoring."""
        # TODO 1: Create execution plan using topological sort from DAG engine
        # Hint: Use create_execution_plan(pipeline, estimated_durations)

        # TODO 2: Initialize task executions in PENDING state
        # Hint: Create TaskExecution for each task using state_manager.create_execution()

        # TODO 3: Start execution levels in dependency order
        # Hint: Process execution_plan.execution_levels sequentially, tasks within level in parallel

        # TODO 4: Monitor running tasks and handle state transitions
        # Hint: Use worker threads to poll task status and call _handle_task_completion()

        # TODO 5: Handle failures and determine if pipeline should continue
        # Hint: Check retry policies, update dependent task states, decide on pipeline failure

        # TODO 6: Clean up resources and persist final state
        # Hint: Release all allocated resources, flush metrics, update pipeline run status

    def _execute_task_level(self, task_ids: List[str], run_id: str,
```

```
                        parameters: Dict[str, Any]) -> Dict[str, bool]:
    """Execute all tasks in a dependency level in parallel."""
    # TODO 1: Check resource availability for all tasks in level
    # Hint: Sum resource requirements and compare to limits

    # TODO 2: Allocate resources and transition tasks to QUEUED state
    # Hint: Reserve DB connections, temp directories, update task state

    # TODO 3: Submit tasks to thread pool for parallel execution
    # Hint: Use ThreadPoolExecutor or ProcessPoolExecutor based on task type

    # TODO 4: Wait for all tasks to complete with timeout
    # Hint: Use concurrent.futures.wait() with appropriate timeout

    # TODO 5: Collect results and handle any exceptions
    # Hint: Check Future.exception() for each completed task

    return {}  # task_id -> success mapping

def _execute_single_task(self, task_def: TaskDefinition, run_id: str,
                        parameters: Dict[str, Any]) -> bool:
    """Execute individual task with monitoring and error handling."""
    task_id = task_def.id
    start_time = time.time()

    # TODO 1: Transition task to RUNNING state
    # Hint: Use state_manager.transition_state() with EXECUTION_STARTED event

    # TODO 2: Load appropriate connector/transformer based on task type
    # Hint: Use factory pattern to instantiate correct handler for task_def.type

    # TODO 3: Execute task with timeout and resource monitoring
    # Hint: Wrap execution in try/except, measure resource usage, enforce timeout
```

```
        # TODO 4: Record execution metrics (duration, rows processed, etc.)
        # Hint: Use metrics.record_metric() for performance and business metrics

        # TODO 5: Handle success/failure and update task state appropriately
        # Hint: Call state_manager.transition_state() with appropriate event

        # TODO 6: Clean up any task-specific resources
        # Hint: Close connections, delete temp files, release memory

        return False  # Return success/failure
```

Scheduler Integration Core:

PYTHON

```python
class CronScheduler:
    """Handles time-based pipeline scheduling with catchup and overlap handling."""

    def __init__(self, execution_engine: TaskExecutionEngine):
        self.execution_engine = execution_engine
        self.registered_pipelines: Dict[str, PipelineDefinition] = {}
        self.active_runs: Dict[str, str] = {}  # pipeline_id -> run_id
        self.running = False

    def start_scheduler(self):
        """Start the scheduler polling loop."""
        self.running = True
        threading.Thread(target=self._scheduler_loop, daemon=True).start()

    def register_pipeline(self, pipeline: PipelineDefinition) -> bool:
        """Register pipeline for scheduled execution."""
        # TODO 1: Validate pipeline definition and schedule expression
        # Hint: Use validate_pipeline() and parse cron expression

        # TODO 2: Check for schedule conflicts with existing pipelines
        # Hint: Evaluate if overlapping schedules might cause resource conflicts

        # TODO 3: Store pipeline in registry with schedule metadata
        # Hint: Include next run time calculation and execution policy

        # TODO 4: Log registration and trigger initial schedule calculation
        # Hint: Calculate next execution time and log for monitoring

        return False

    def _scheduler_loop(self):
        """Main scheduler polling loop that evaluates and triggers pipelines."""
        while self.running:
```

```
            # TODO 1: Calculate current time and evaluation window
            # Hint: Use timezone-aware datetime, consider scheduling tolerance window

            # TODO 2: Evaluate each registered pipeline for execution readiness
            # Hint: Check if current time matches cron expression, handle missed executions

            # TODO 3: Apply execution policies for ready pipelines
            # Hint: Check ALLOW_CONCURRENT, SKIP_ON_RUNNING, etc. policies

            # TODO 4: Trigger pipeline executions and track active runs
            # Hint: Generate run_id, call execution_engine.execute_pipeline(), update tracking

            # TODO 5: Clean up completed runs and update metrics
            # Hint: Remove from active_runs, record scheduling metrics

            # TODO 6: Sleep until next evaluation cycle
            # Hint: Use configurable poll interval, typically 30-60 seconds

            time.sleep(30)  # Default poll interval
```

Milestone Checkpoint

After implementing the orchestration and monitoring components, verify the system with these checkpoints:

Test Command:

```bash
python -m pytest tests/test_orchestration.py -v

python scripts/run_sample_pipeline.py --pipeline simple_etl --monitor true
```

Expected Behavior:

1. **Scheduler Registration**: Register a simple pipeline with 5-minute cron schedule, verify next execution time calculation
2. **Task Execution**: Execute pipeline manually, observe parallel task execution within dependency levels
3. **State Transitions**: Monitor task state changes from PENDING → QUEUED → RUNNING → SUCCESS
4. **Metrics Collection**: Verify metrics are collected for task duration, success rate, and resource usage
5. **Alert Generation**: Trigger a task failure and verify alert is generated with appropriate severity
6. **Resource Management**: Execute multiple pipelines simultaneously and verify resource limits are respected

Verification Steps:

```bash
# Check scheduler registration
curl http://localhost:8080/api/schedules | jq .

# Monitor pipeline execution
curl http://localhost:8080/api/runs/active | jq .

# View collected metrics
sqlite3 metrics.db "SELECT * FROM task_metrics ORDER BY timestamp DESC LIMIT 10;"

# Test alerting
curl -X POST http://localhost:8080/api/test/trigger-failure
```

Signs of Issues:

- Tasks stuck in QUEUED state: Check resource availability and thread pool configuration
- Missing metrics: Verify metrics collector background flush is running
- Alert flooding: Check alert suppression logic and severity classification
- Memory leaks during long runs: Verify resource cleanup in finally blocks

Interactions and Data Flow

> **Milestone(s):** All milestones - describes how components from pipeline definition (Milestone 1), data processing (Milestones 2-3), and orchestration (Milestone 4) interact through well-defined interfaces and protocols.

Mental Model: Orchestra Conductor

Think of the ETL system as a symphony orchestra performing a complex musical piece. The **DAG Definition Engine** is like the sheet music - it defines what needs to be played, when, and in what order. The **Pipeline Orchestration Engine** acts as the conductor, coordinating timing, cueing different sections, and managing the overall flow. Individual **task executors** are like musicians - each with specialized skills (violinists, trumpeters, percussionists) who execute their parts when signaled. The **monitoring system** is like the audience and recording equipment - observing the performance, capturing what happened, and providing feedback. Just as musicians must follow precise timing and hand-off cues between sections, ETL components communicate through well-defined message protocols and data contracts.

The beauty of this orchestration lies in the coordination - when the conductor raises their baton to start the performance, a cascade of precisely-timed interactions begins. First movement starts with strings (extraction tasks), then woodwinds join in (transformation tasks), finally brass completes the harmony (loading tasks). Each musician knows exactly when to play their part based on visual cues from the conductor and audio cues from other sections. Similarly, each ETL component knows when to execute based on state transitions and dependency signals from other components.

Component Communication

The ETL system components communicate through a combination of **synchronous API calls** for control operations and **asynchronous message passing** for data processing events. This hybrid approach balances the need for immediate feedback on critical operations with the scalability requirements of high-throughput data processing.

Decision: Hybrid Synchronous/Asynchronous Communication

- **Context**: Components need both immediate control feedback and scalable event processing
- **Options Considered**: Pure synchronous APIs, pure message queues, hybrid approach
- **Decision**: Hybrid synchronous APIs for control, asynchronous messages for events
- **Rationale**: Control operations need immediate success/failure feedback, but data events need decoupling for scalability
- **Consequences**: Enables responsive control plane while maintaining scalable data plane, but requires managing two communication patterns

Communication Type	Use Cases	Pattern	Benefits	Trade-offs
Synchronous APIs	Pipeline registration, schedule updates, manual triggers	Request/Response	Immediate feedback, simple error handling	Blocking operations, limited scalability
Asynchronous Messages	Task state changes, metrics collection, lineage events	Pub/Sub	High throughput, loose coupling	Eventual consistency, complex error handling
Shared State	Task execution status, pipeline run metadata	Database/Cache	Persistent state, complex queries	Potential bottleneck, consistency complexity

Control Plane APIs

The **control plane** handles pipeline management operations through RESTful APIs. These operations require immediate feedback and typically have lower frequency but higher reliability requirements.

Method	Endpoint	Request Format	Response Format	Description
`register_pipeline`	POST /api/v1/pipelines	`PipelineDefinition`	`{"pipeline_id": str, "version": int}`	Register new pipeline definition
`update_pipeline`	PUT /api/v1/pipelines/{id}	`PipelineDefinition`	`{"pipeline_id": str, "version": int}`	Update existing pipeline
`trigger_pipeline`	POST /api/v1/pipelines/{id}/runs	`{"parameters": dict, "priority": int}`	`{"run_id": str, "status": str}`	Manually trigger pipeline execution
`get_pipeline_status`	GET /api/v1/pipelines/{id}/status	None	`{"status": str, "last_run": datetime, "next_run": datetime}`	Retrieve pipeline execution status
`pause_pipeline`	POST /api/v1/pipelines/{id}/pause	`{"reason": str}`	`{"success": bool}`	Temporarily disable pipeline
`get_task_logs`	GET /api/v1/runs/{run_id}/tasks/{task_id}/logs	None	`{"logs": List[str], "metrics": dict}`	Retrieve task execution logs

Each API endpoint follows a standard request/response pattern with consistent error handling. All endpoints return HTTP status codes that map directly to operation outcomes: 200 for success, 400 for validation errors, 404 for missing resources, 500 for system errors. Error responses include structured error objects with error codes, human-readable messages, and context information for debugging.

Data Plane Messaging

The **data plane** handles high-frequency operational events through asynchronous messaging. This system uses a publish/subscribe pattern where components publish events to named topics and subscribe to events they need to process.

Event Type	Topic	Message Format	Publisher	Subscribers
Task State Change	`task.state.{pipeline_id}`	`TaskStateEvent`	Task Executor	Orchestrator, Monitoring
Data Lineage	`lineage.{pipeline_id}`	`LineageEvent`	Connectors, Transformers	Lineage Tracker
Metrics Collection	`metrics.{component}`	`MetricsEvent`	All Components	Monitoring Dashboard
Pipeline Progress	`progress.{run_id}`	`ProgressEvent`	Orchestrator	UI, Alerting
Resource Usage	`resources.{executor_id}`	`ResourceEvent`	Task Executors	Resource Manager

Message Format Specifications:

Field	Type	Description	Required	Example
`event_id`	str	Unique event identifier	Yes	"evt_123e4567-e89b-12d3"
`timestamp`	datetime	Event occurrence time	Yes	"2024-01-15T14:30:00Z"
`source_component`	str	Component that generated event	Yes	"task_executor_3"
`event_type`	str	Specific event classification	Yes	"TASK_STATE_CHANGED"
`pipeline_id`	str	Associated pipeline identifier	Yes	"customer_data_pipeline"
`run_id`	str	Associated pipeline run	Yes	"run_20240115_143000"
`task_id`	str	Associated task identifier	No	"extract_customer_data"
`payload`	dict	Event-specific data	Yes	{"old_state": "RUNNING", "new_state": "SUCCESS"}
`correlation_id`	str	Request correlation identifier	No	"req_987fcdeb-51a2-34b5"

Inter-Component State Synchronization

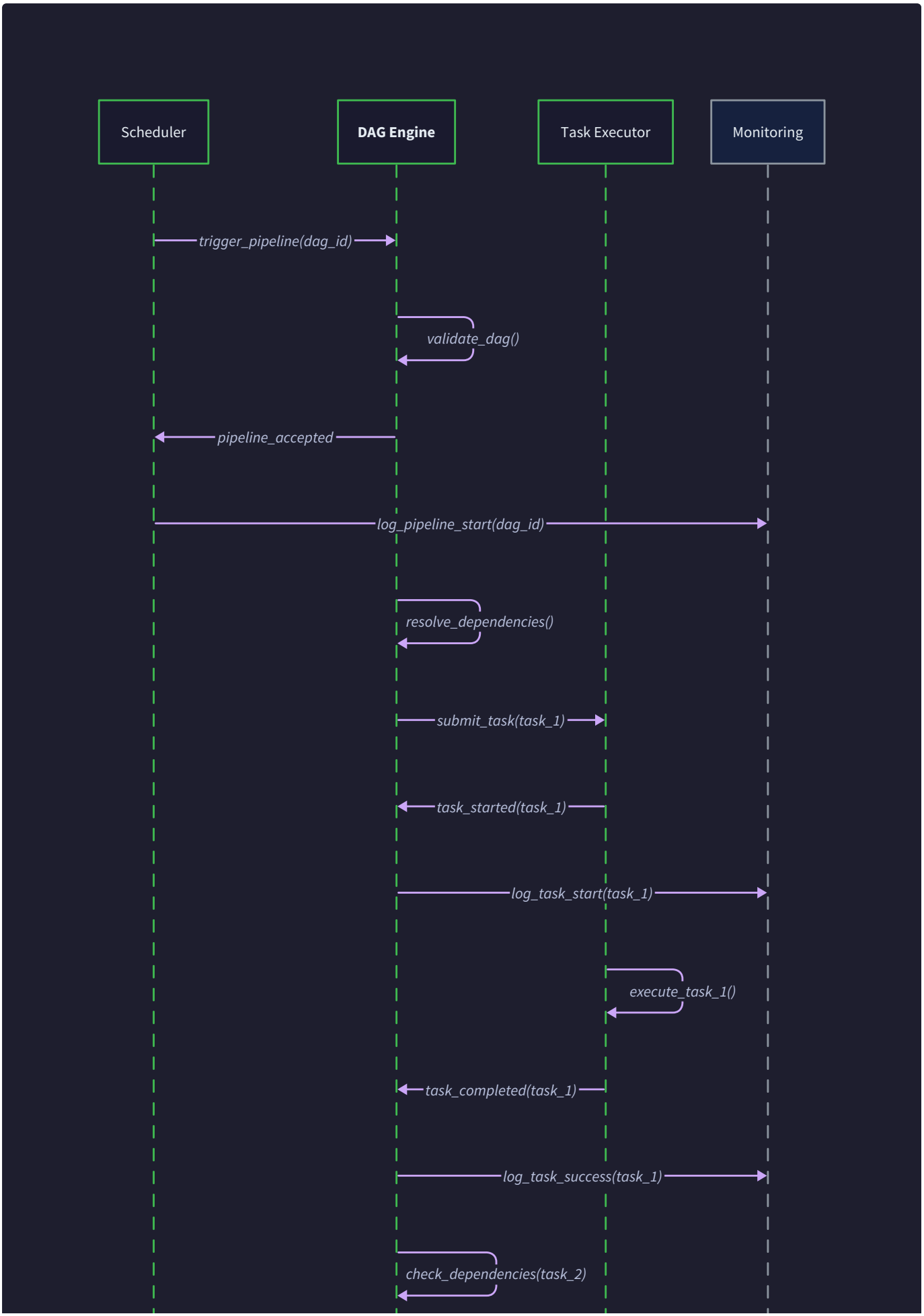
Components maintain consistency through a combination of **authoritative state ownership** and **event-driven synchronization**. Each component owns specific state domains and publishes changes as events that other components can consume to maintain their derived views.

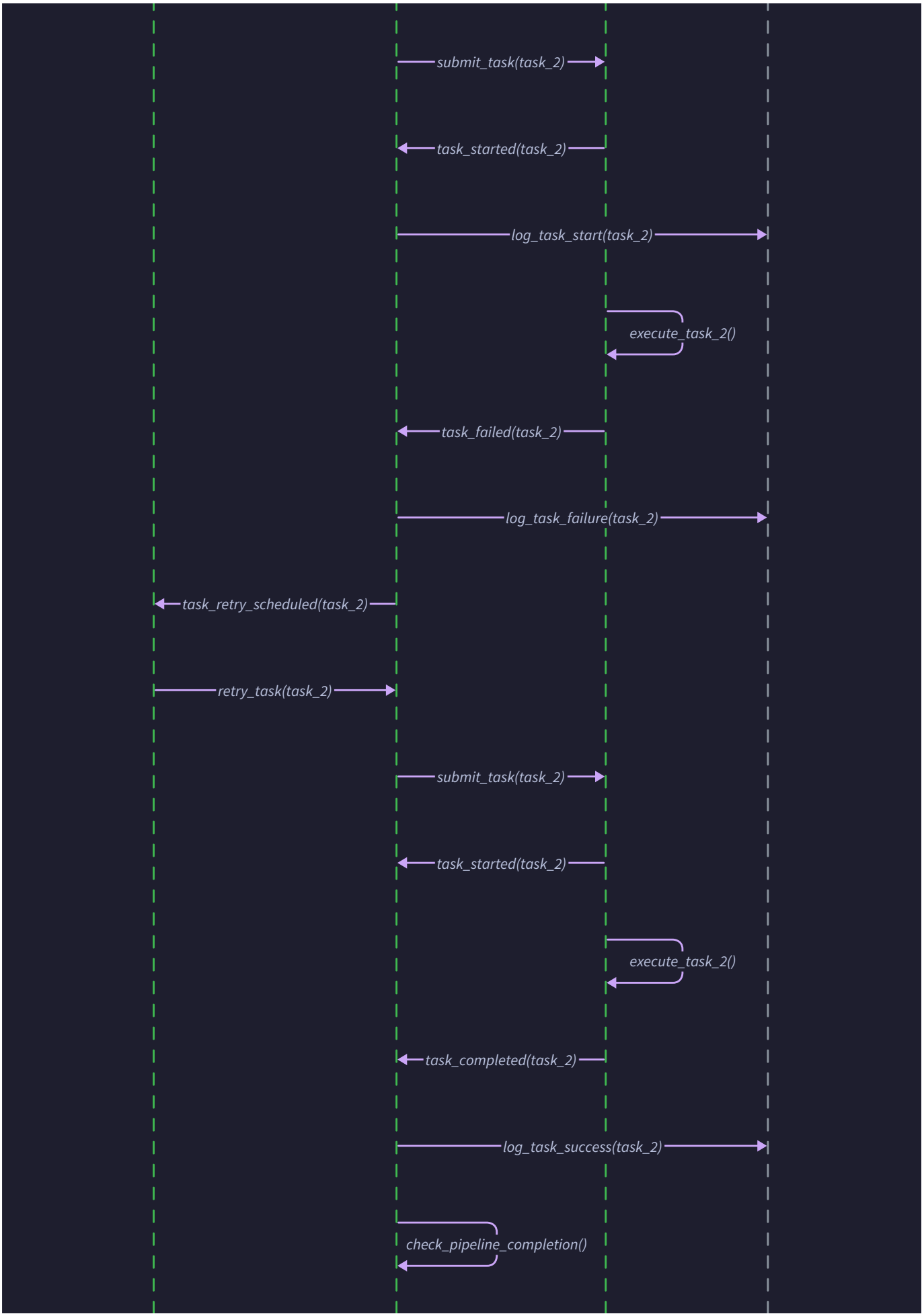
> **Key Design Principle**: State ownership is clearly partitioned - the Orchestrator owns pipeline run state, Task Executors own task execution state, and the Monitoring system owns aggregated metrics. No component directly modifies another component's authoritative state.

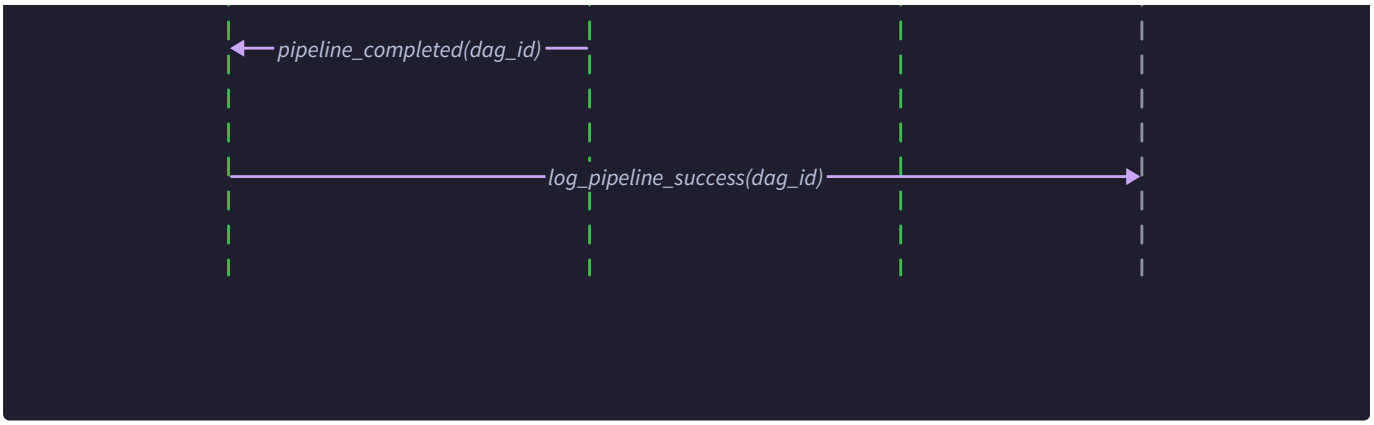
Component	Owned State	Published Events	Subscribed Events
DAG Engine	Pipeline definitions, validation results	`pipeline.registered`, `dag.validated`	None
Orchestrator	Pipeline runs, execution plans	`run.started`, `run.completed`, `task.scheduled`	`task.state.*`, `executor.heartbeat`
Task Executor	Task executions, resource usage	`task.state.*`, `metrics.executor`	`task.scheduled`, `run.cancelled`
Scheduler	Schedule configurations, next run times	`schedule.triggered`, `schedule.updated`	`run.completed`, `pipeline.updated`
Monitoring	Aggregated metrics, alerts	`alert.triggered`, `metrics.aggregated`	`task.state.*`, `metrics.*`

Pipeline Execution Flow

A complete pipeline execution involves a carefully orchestrated sequence of interactions between components. Understanding this flow is crucial for debugging issues and optimizing performance.


Scheduler
DAG Engine
Task Executor
Monitoring
trigger_pipeline(dag_id)
validate_dag()
pipeline_accepted
log_pipeline_start(dag_id)
resolve_dependencies()
submit_task(task_1)
task_started(task_1)
log_task_start(task_1)
execute_task_1()
task_completed(task_1)
log_task_success(task_1)
check_dependencies(task_2)


submit_task(task_2)
task_started(task_2)
log_task_start(task_2)
execute_task_2()
task_failed(task_2)
log_task_failure(task_2)
task_retry_scheduled(task_2)
retry_task(task_2)
submit_task(task_2)
task_started(task_2)
execute_task_2()
task_completed(task_2)
log_task_success(task_2)
check_pipeline_completion()



Phase 1: Pipeline Trigger and Validation

The execution flow begins when the **Scheduler** determines that a pipeline should run based on its schedule configuration or when a manual trigger is received through the control API.

1. **Schedule Evaluation**: The Scheduler evaluates all active pipeline schedules using their cron expressions and determines which pipelines are ready to execute. It checks the current time against the `get_next_run_time()` calculation for each pipeline.
2. **Trigger Generation**: When a pipeline's execution time arrives, the Scheduler generates a trigger event containing the pipeline ID, scheduled execution time, and any default parameters defined in the schedule configuration.
3. **Trigger Validation**: The Orchestrator receives the trigger event and performs initial validation checks. It verifies that the pipeline definition exists, is not paused, and that the execution policy allows a new run (checking concurrent execution rules).
4. **Run Instance Creation**: The Orchestrator creates a new pipeline run instance with a unique `run_id`, initial state of `INITIALIZING`, and timestamps for tracking. It also resolves any runtime parameters by merging scheduled parameters with pipeline defaults.
5. **DAG Retrieval and Validation**: The Orchestrator calls `get_pipeline(pipeline_id)` to retrieve the current pipeline definition, then invokes `validate_pipeline(pipeline)` to ensure the DAG is still valid (no cycles, all task definitions complete).
6. **Execution Plan Generation**: The Orchestrator calls `create_execution_plan(pipeline, task_durations)` to determine the optimal execution order, parallel execution levels, and resource requirements. This plan guides the entire execution process.

> **Critical Insight**: The execution plan is generated fresh for each run, allowing the system to incorporate updated task duration estimates and current resource availability. This dynamic planning enables better resource utilization than static execution orders.

Phase 2: Task Scheduling and Dependency Resolution

Once the execution plan is ready, the Orchestrator begins scheduling individual tasks based on their dependency relationships and resource requirements.

7. **Initial Task Identification**: The Orchestrator examines the execution plan's first level (tasks with no dependencies) and marks them as eligible for scheduling. These tasks transition from `PENDING` to `WAITING` state.
8. **Resource Availability Check**: For each eligible task, the Orchestrator queries available Task Executors to find those with sufficient resources (CPU, memory, storage) to handle the task based on its resource requirements specification.
9. **Task Assignment**: The Orchestrator assigns tasks to available executors using a load balancing algorithm that considers current executor load, task resource requirements, and data locality hints. Each assignment creates a `TaskExecution` record with initial state `QUEUED`.
10. **Execution Message Dispatch**: The Orchestrator publishes task execution messages to the `task.scheduled` topic, containing task definitions, runtime parameters, and execution context. Task Executors subscribe to these messages and begin processing assigned tasks.

11. **Dependency Tracking**: The Orchestrator maintains a dependency tracking matrix that monitors which tasks are running, completed, or failed. This matrix enables efficient determination of when downstream tasks become eligible for execution.

Phase 3: Task Execution and State Management

Task Executors receive task assignments and manage the actual execution of extraction, transformation, and loading operations.

12. **Task Initialization**: When a Task Executor receives a task assignment, it transitions the task state to `RUNNING` and publishes a state change event. It also allocates local resources and establishes any required connections to data sources.
13. **Data Processing Execution**: The executor invokes the appropriate connector or transformer based on the task type. For extraction tasks, it calls `extract(query, options)` to retrieve data. For transformation tasks, it calls transformation functions. For loading tasks, it calls `load(data_stream, target, options)`.
14. **Progress Monitoring**: During execution, the Task Executor periodically publishes progress events containing metrics like records processed, bytes transferred, and execution time. These events enable real-time monitoring and early detection of performance issues.
15. **Error Handling and Retries**: If a task encounters an error, the executor evaluates the task's `RetryPolicy` to determine whether to retry immediately, schedule a delayed retry with exponential backoff, or mark the task as failed. Each decision triggers appropriate state transitions.
16. **Task Completion**: Upon successful completion, the executor publishes a task completion event with final metrics and output metadata. It also performs cleanup of local resources and temporary files.

Phase 4: Dependency Propagation and Pipeline Completion

As tasks complete, the Orchestrator updates its dependency tracking and schedules downstream tasks that become eligible for execution.

17. **Dependency Update**: When the Orchestrator receives a task completion event, it updates its dependency matrix and identifies downstream tasks whose dependencies are now satisfied. These tasks transition from `PENDING` to `WAITING` state.
18. **Next Level Scheduling**: The Orchestrator repeats the task scheduling process (steps 8-11) for newly eligible tasks, maintaining the parallel execution levels defined in the execution plan while respecting resource constraints.
19. **Pipeline Progress Tracking**: The Orchestrator continuously tracks overall pipeline progress by monitoring the completion ratio of tasks at each execution level. It publishes pipeline progress events that external systems can consume for dashboards and notifications.
20. **Failure Impact Analysis**: If any task fails and exhausts its retry attempts, the Orchestrator analyzes the impact on downstream tasks. Depending on the pipeline's failure policy, it may cancel dependent tasks, mark them as skipped, or continue execution of independent task branches.
21. **Pipeline Completion**: The pipeline completes when all tasks have reached terminal states (SUCCESS, FAILED, or SKIPPED). The Orchestrator calculates final pipeline status, aggregates metrics from all tasks, and publishes a pipeline completion event.

Phase 5: Cleanup and Lineage Recording

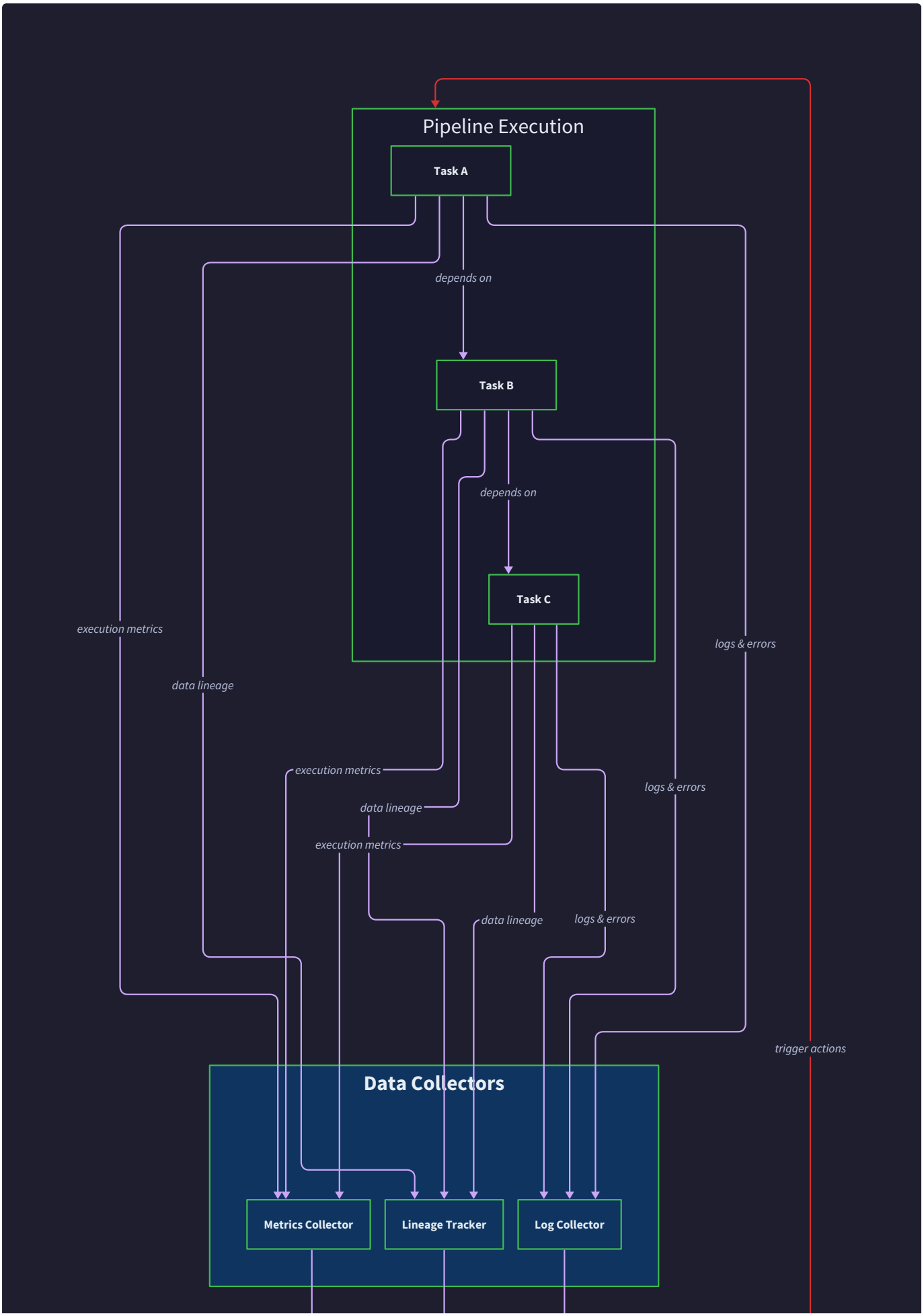
The final phase ensures proper resource cleanup and captures complete data lineage information for audit and debugging purposes.

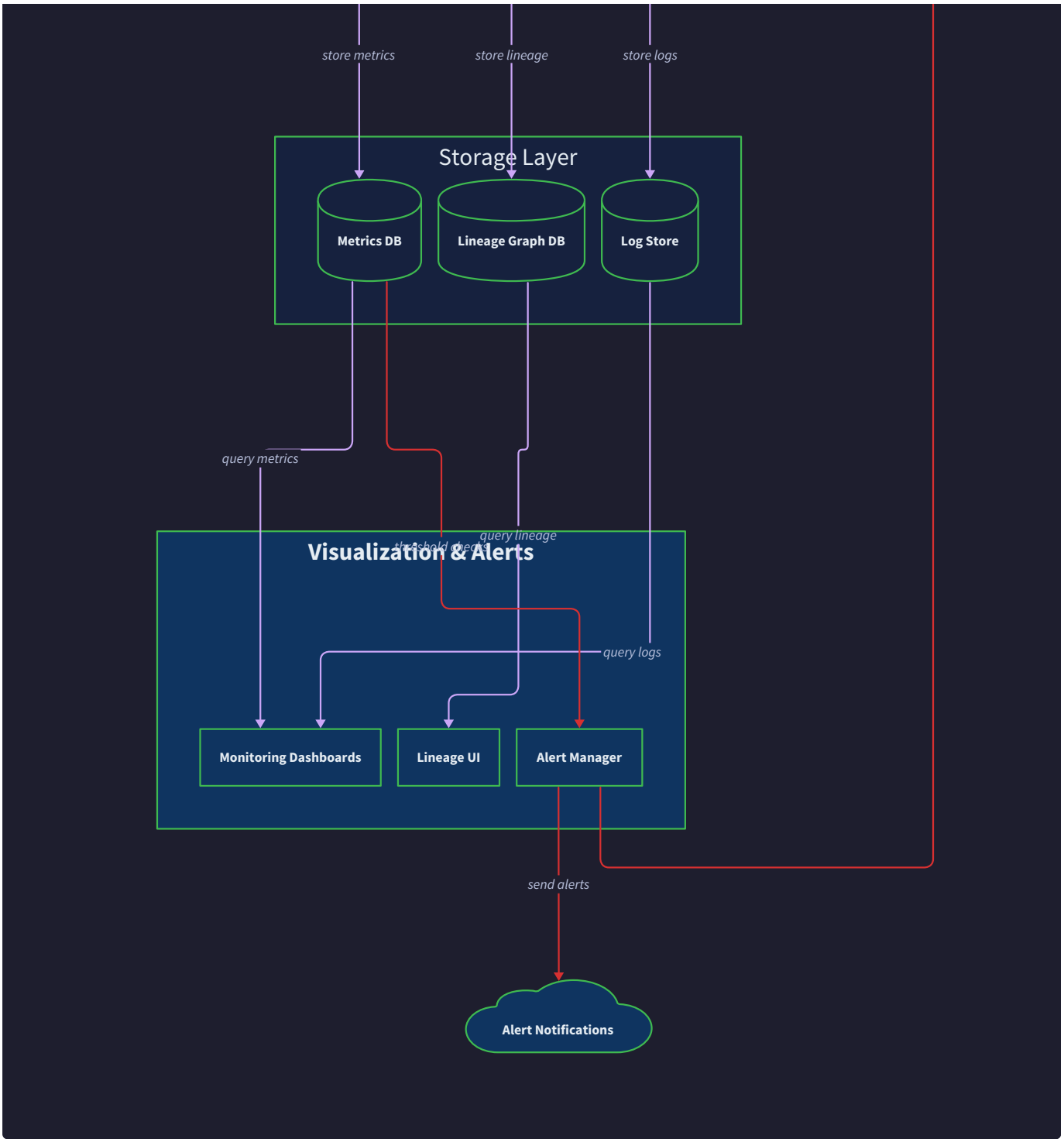
22. **Resource Cleanup**: Task Executors perform final cleanup of any resources allocated for the pipeline run, including temporary files, database connections, and allocated memory. They also publish resource deallocation events for capacity planning.
23. **Lineage Consolidation**: The Lineage Tracker aggregates all lineage events published during the pipeline run to create a complete data provenance record. This includes source data locations, transformation operations applied, and destination data locations.
24. **Metrics Aggregation**: The Monitoring system calculates pipeline-level metrics by aggregating task-level metrics. This includes total execution time, data volume processed, resource utilization, and error rates.

25. **Audit Log Creation**: The system generates comprehensive audit logs containing the complete execution history, all state transitions, error messages, and performance metrics. These logs are stored for compliance and debugging purposes.
26. **Schedule Update**: The Scheduler updates the pipeline's next execution time based on its cron schedule and records the completion of the current run. This ensures proper spacing of future executions according to the schedule configuration.

Data Lineage Tracking

Data lineage provides a complete audit trail of how data flows through the ETL pipeline, enabling data governance, impact analysis, and debugging of data quality issues. The lineage system captures not just what data was processed, but how it was transformed and where it ended up.


Pipeline Execution
Task A
depends on
Task B
depends on
Task C
execution metrics
data lineage
logs & errors
execution metrics
data lineage
logs & errors
execution metrics
data lineage
logs & errors
trigger actions
Data Collectors
Metrics Collector
Lineage Tracker
Log Collector



Mental Model: Evidence Chain in Investigation

Think of data lineage like the chain of evidence in a criminal investigation. Every piece of evidence must be tracked from its original location through every person who handled it, every test performed on it, and every conclusion drawn from it. If a piece of evidence becomes contaminated or questions arise about its authenticity, investigators can trace back through the complete chain to identify where problems occurred. Similarly, when data quality issues arise in production reports, data lineage allows you to trace back through every transformation, join, and aggregation to identify the root cause.

Just as evidence must be handled by authorized personnel following documented procedures, data in the ETL pipeline should only be modified by authorized transformation steps following defined business rules. The lineage system acts like the evidence log book, recording every hand-off and every operation performed.

Lineage Data Model

The lineage system tracks relationships between **data assets** (databases, tables, files), **transformations** (ETL tasks), and **pipeline runs** through a graph-based model that captures both schema-level and instance-level lineage.

Field	Type	Description	Example
`lineage_id`	str	Unique identifier for lineage record	"lineage_123e4567-e89b"
`pipeline_run_id`	str	Associated pipeline execution	"run_20240115_143000"
`task_id`	str	Task that performed transformation	"transform_customer_data"
`source_assets`	List[AssetReference]	Input data assets consumed	[{"type": "table", "name": "raw.customers"}]
`target_assets`	List[AssetReference]	Output data assets produced	[{"type": "table", "name": "clean.customers"}]
`transformation_type`	str	Type of operation performed	"COLUMN_MAPPING", "AGGREGATION", "JOIN"
`transformation_logic`	str	SQL query or transformation code	"SELECT customer_id, UPPER(name) FROM raw.customers"
`schema_changes`	List[SchemaChange]	Column additions, deletions, renames	[{"type": "RENAME", "from": "cust_name", "to": "customer_name"}]
`data_profile`	dict	Statistical summary of processed data	{"row_count": 1000000, "null_percentage": 0.02}
`execution_timestamp`	datetime	When transformation occurred	"2024-01-15T14:35:22Z"
`lineage_metadata`	dict	Additional context and annotations	{"business_purpose": "data cleanup", "data_steward": "alice"}

AssetReference Structure:

Field	Type	Description	Example
`asset_type`	str	Type of data asset	"database_table", "file", "api_endpoint"
`asset_name`	str	Fully qualified asset name	"warehouse.clean.customers"
`asset_schema`	str	Schema or namespace	"clean"
`location_uri`	str	Physical location or connection string	"postgresql://warehouse:5432/clean/customers"
`partition_info`	dict	Partition or file path details	{"date_partition": "2024-01-15", "file_path": "/data/customers/2024/01/15/"}
`column_lineage`	List[ColumnLineage]	Field-level transformation details	[{"source_column": "cust_name", "target_column": "customer_name", "transformation": "UPPER()"}]

Lineage Collection Strategy

The ETL system employs a **multi-level lineage collection** strategy that captures lineage information at different granularities depending on the transformation complexity and business requirements.

Decision: Multi-Level Lineage Collection

- **Context**: Different stakeholders need lineage at different levels of detail
- **Options Considered**: Table-level only, column-level only, configurable multi-level
- **Decision**: Implement configurable multi-level collection with table, column, and value-level tracking
- **Rationale**: Business users need table-level for impact analysis, data stewards need column-level for compliance, developers need value-level for debugging
- **Consequences**: Enables flexible lineage reporting but increases storage and processing overhead

Lineage Collection Levels:

Level	Granularity	Collection Method	Use Cases	Performance Impact
Dataset	Table/file level	Connector metadata	Impact analysis, data discovery	Minimal
Schema	Column level	SQL parsing, schema inference	Compliance reporting, data mapping	Low
Instance	Row/record level	Data sampling, checksums	Data quality debugging, auditing	High
Value	Field transformation level	Complete data capture	Regulatory compliance, forensics	Very High

Automatic Lineage Capture

The lineage system automatically captures lineage information during pipeline execution without requiring explicit instrumentation from pipeline developers. This automatic collection uses multiple techniques depending on the transformation type.

SQL-Based Transformation Lineage:

For SQL transformations, the system uses **SQL parsing and analysis** to extract lineage relationships automatically. The SQL parser analyzes SELECT statements to identify source tables, join relationships, filter conditions, and column transformations.

1. **Query Parsing**: The transformation engine intercepts SQL queries before execution and parses them using a SQL abstract syntax tree (AST) parser that understands the specific SQL dialect being used.
2. **Dependency Extraction**: The parser identifies all referenced tables, views, and columns in FROM, JOIN, and WHERE clauses, creating source asset references with exact column mappings.
3. **Transformation Logic Capture**: The parser extracts column expressions from the SELECT clause, including function calls, arithmetic operations, and case statements, preserving the exact transformation logic applied.
4. **Output Schema Inference**: The parser predicts output column names and types based on the SELECT clause analysis, creating target asset references that match the actual query results.

Python UDF Lineage:

For Python user-defined functions, the system uses **code instrumentation and runtime inspection** to capture lineage relationships that cannot be determined through static analysis.

1. **Function Registration**: When UDFs are registered using `register_function(name, func)`, the system wraps the function with lineage collection decorators that intercept input and output data.
2. **Input Tracking**: The wrapper captures metadata about input data streams, including schema information, data source references, and statistical profiles of the input data.
3. **Output Analysis**: The wrapper analyzes function outputs to determine output schema, data transformations applied, and relationships between input and output fields.
4. **Manual Annotations**: The system provides decorators that allow UDF developers to explicitly declare lineage relationships when automatic detection is insufficient, such as complex business logic or external API calls.

Lineage Query and Analysis

The lineage system provides both programmatic APIs and query interfaces for analyzing data provenance and impact relationships. These capabilities support both interactive exploration and automated governance workflows.

Forward Impact Analysis:

Forward impact analysis answers the question: "If this source data changes, what downstream systems will be affected?" This analysis is crucial for change management and data governance.

Query Type	Method	Parameters	Returns	Use Case
Direct Dependencies	`get_direct_dependencies(asset)`	Asset reference	List of immediate downstream assets	Change impact assessment
Transitive Dependencies	`get_all_dependencies(asset, max_depth)`	Asset reference, traversal depth	Complete dependency graph	Full impact analysis
Pipeline Impact	`get_affected_pipelines(asset)`	Asset reference	List of pipeline IDs	Pipeline scheduling decisions
Schema Impact	`get_schema_dependencies(asset, column)`	Asset and column references	Column-level dependency graph	Schema evolution planning

Backward Provenance Analysis:

Backward provenance analysis answers: "Where did this data come from and how was it transformed?" This analysis supports data quality investigations and audit requirements.

Query Type	Method	Parameters	Returns	Use Case
Data Sources	`get_data_sources(asset)`	Asset reference	List of source assets and transformations	Root cause analysis
Transformation History	`get_transformation_path(source, target)`	Source and target assets	Complete transformation sequence	Data quality debugging
Business Logic Trace	`get_business_rules(asset, column)`	Asset and column references	Applied business rules and transformations	Compliance auditing
Temporal Lineage	`get_lineage_at_time(asset, timestamp)`	Asset reference and time	Historical lineage relationships	Point-in-time analysis

Common Pitfalls

⚠️ **Pitfall: Lineage Collection Overhead** Many implementations collect too much lineage information by default, severely impacting pipeline performance. Collecting row-level lineage for high-volume data streams can slow down pipelines by 50% or more. **Solution**: Use tiered collection levels and sample-based collection for high-volume streams. Configure collection levels based on data sensitivity and regulatory requirements.

⚠️ **Pitfall: Incomplete Schema Lineage** SQL parsing often misses complex transformations like UDF calls, dynamic SQL, or external API enrichments, creating gaps in lineage graphs. **Solution**: Implement hybrid collection that combines automatic SQL analysis with manual lineage annotations for complex transformations. Provide clear documentation on when manual annotations are required.

⚠️ **Pitfall: Lineage Storage Explosion** Storing complete lineage for every pipeline run can quickly consume massive storage, especially for high-frequency pipelines processing large datasets. **Solution**: Implement lineage retention policies that keep detailed lineage for recent runs but aggregate older lineage to schema-level relationships. Consider compressing lineage for archived pipeline runs.

⚠️ **Pitfall: Cross-System Lineage Gaps** Lineage often breaks when data moves between different systems (ETL → data warehouse → BI tools) because each system maintains its own lineage model. **Solution**: Standardize on lineage metadata formats and implement lineage bridges that can import/export lineage information between systems. Use standard formats like OpenLineage for interoperability.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Message Broker	Redis Streams with Python redis-py	Apache Kafka with confluent-kafka-python
API Framework	FastAPI with Pydantic validation	FastAPI + Celery for async processing
State Storage	SQLite with SQLAlchemy ORM	PostgreSQL with asyncpg for performance
Lineage Storage	JSON documents in primary database	Graph database (Neo4j) with py2neo
Monitoring	Python logging + Prometheus client	OpenTelemetry with Jaeger tracing

Recommended File Structure

```
etl_pipeline/
  core/
    communication/
      __init__.py
      message_broker.py          ← Message publishing and subscription
      api_server.py              ← REST API endpoints
      event_handlers.py          ← Async event processing
    orchestration/
      __init__.py
      pipeline_executor.py       ← Main execution flow coordination
      dependency_tracker.py      ← Task dependency resolution
      state_manager.py           ← Pipeline and task state management
    lineage/
      __init__.py
      lineage_collector.py       ← Automatic lineage capture
      lineage_analyzer.py        ← Forward/backward analysis
      sql_parser.py             ← SQL lineage extraction
  examples/
    sample_pipeline.py           ← Complete end-to-end example
    lineage_queries.py          ← Example lineage analysis queries
  tests/
    integration/
      test_pipeline_execution.py ← Full pipeline execution tests
      test_lineage_tracking.py   ← End-to-end lineage tests
```

Infrastructure Starter Code

Message Broker Implementation:

PYTHON

```python
"""
Message broker implementation for component communication.
Provides both synchronous API calls and asynchronous event publishing.
"""
import json
import asyncio
from typing import Dict, List, Callable, Any, Optional
from dataclasses import dataclass, asdict
from datetime import datetime
import redis
import logging

@dataclass
class MessageEvent:
    """Standard message format for all inter-component communication."""
    event_id: str
    timestamp: datetime
    source_component: str
    event_type: str
    pipeline_id: str
    run_id: str
    task_id: Optional[str] = None
    payload: Dict[str, Any] = None
    correlation_id: Optional[str] = None

class MessageBroker:
    """
    Handles all inter-component messaging for the ETL system.
    Supports both pub/sub patterns for events and point-to-point for control messages.
    """


    def __init__(self, redis_url: str = "redis://localhost:6379"):
        self.redis_client = redis.Redis.from_url(redis_url, decode_responses=True)
        self.subscribers: Dict[str, List[Callable]] = {}
```

```python
        self.logger = logging.getLogger(__name__)


    async def publish_event(self, topic: str, event: MessageEvent) -> bool:
        """
        Publish event to specified topic for subscriber consumption.
        Returns True if published successfully, False otherwise.
        """
        try:
            message_data = json.dumps(asdict(event), default=str)
            result = self.redis_client.xadd(topic, {"event": message_data})
            self.logger.debug(f"Published event {event.event_id} to topic {topic}")
            return result is not None
        except Exception as e:
            self.logger.error(f"Failed to publish event {event.event_id}: {e}")
            return False


    def subscribe_to_topic(self, topic: str, handler: Callable[[MessageEvent], None]) -> None:
        """Register event handler for specified topic."""
        if topic not in self.subscribers:
            self.subscribers[topic] = []
        self.subscribers[topic].append(handler)
        self.logger.info(f"Registered handler for topic {topic}")


    async def start_consumer(self, consumer_group: str, consumer_name: str) -> None:
        """Start consuming messages from subscribed topics."""
        while True:
            try:
                for topic in self.subscribers.keys():
                    # Create consumer group if it doesn't exist
                    try:
                        self.redis_client.xgroup_create(topic, consumer_group, id='0', mkstream=True)
                    except redis.ResponseError:
                        pass  # Group already exists
```

```
            # Read messages from the stream
            messages = self.redis_client.xreadgroup(
                consumer_group, consumer_name, {topic: '>'}, count=10, block=1000
            )

            for stream, msgs in messages:
                for msg_id, fields in msgs:
                    try:
                        event_data = json.loads(fields['event'])
                        event = MessageEvent(**event_data)

                        # Call all registered handlers for this topic
                        for handler in self.subscribers.get(topic, []):
                            handler(event)

                        # Acknowledge message processing
                        self.redis_client.xack(topic, consumer_group, msg_id)

                    except Exception as e:
                        self.logger.error(f"Failed to process message {msg_id}: {e}")

        except Exception as e:
            self.logger.error(f"Consumer error: {e}")
            await asyncio.sleep(1)
```

State Management Implementation:

PYTHON

```python
"""
Centralized state management for pipeline and task execution tracking.
Handles state transitions, persistence, and consistency across components.
"""
from typing import Dict, List, Optional, Set
from dataclasses import dataclass
from datetime import datetime
from enum import Enum
import threading
import sqlite3
import json
import logging

class TaskState(Enum):
    PENDING = "PENDING"
    WAITING = "WAITING"
    QUEUED = "QUEUED"
    RUNNING = "RUNNING"
    SUCCESS = "SUCCESS"
    FAILED = "FAILED"
    RETRYING = "RETRYING"
    CANCELLED = "CANCELLED"
    SKIPPED = "SKIPPED"

class TaskEvent(Enum):
    DEPENDENCIES_MET = "DEPENDENCIES_MET"
    EXECUTION_STARTED = "EXECUTION_STARTED"
    EXECUTION_COMPLETED = "EXECUTION_COMPLETED"
    EXECUTION_FAILED = "EXECUTION_FAILED"
    RETRY_SCHEDULED = "RETRY_SCHEDULED"
    MAX_RETRIES_EXCEEDED = "MAX_RETRIES_EXCEEDED"
    CANCELLED_BY_USER = "CANCELLED_BY_USER"
    UPSTREAM_FAILED = "UPSTREAM_FAILED"
```

```
# State transition mapping - defines valid state changes
TRANSITIONS = {
    TaskState.PENDING: {
        TaskEvent.DEPENDENCIES_MET: TaskState.WAITING,
        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED,
        TaskEvent.UPSTREAM_FAILED: TaskState.SKIPPED
    },
    TaskState.WAITING: {
        TaskEvent.EXECUTION_STARTED: TaskState.RUNNING,
        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED,
        TaskEvent.UPSTREAM_FAILED: TaskState.SKIPPED
    },
    TaskState.RUNNING: {
        TaskEvent.EXECUTION_COMPLETED: TaskState.SUCCESS,
        TaskEvent.EXECUTION_FAILED: TaskState.RETRYING,
        TaskEvent.MAX_RETRIES_EXCEEDED: TaskState.FAILED,
        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED
    },
    TaskState.RETRYING: {
        TaskEvent.EXECUTION_STARTED: TaskState.RUNNING,
        TaskEvent.MAX_RETRIES_EXCEEDED: TaskState.FAILED,
        TaskEvent.CANCELLED_BY_USER: TaskState.CANCELLED
    }
}

@dataclass
class TaskExecution:
    """Represents a single task execution within a pipeline run."""
    task_id: str
    pipeline_run_id: str
    state: TaskState
    attempt_count: int
    started_at: Optional[datetime] = None
```

```python
    completed_at: Optional[datetime] = None
    error_message: Optional[str] = None
    logs: List[str] = None
    metrics: Dict[str, float] = None

class StateManager:
    """
    Thread-safe state management for pipeline and task executions.
    Provides atomic state transitions and persistent storage.
    """

    def __init__(self, db_path: str = "etl_state.db"):
        self.db_path = db_path
        self.lock = threading.RLock()
        self.logger = logging.getLogger(__name__)
        self._init_database()

    def _init_database(self):
        """Initialize SQLite database with required tables."""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute("""
            CREATE TABLE IF NOT EXISTS task_executions (
                task_id TEXT,
                pipeline_run_id TEXT,
                state TEXT,
                attempt_count INTEGER,
                started_at TEXT,
                completed_at TEXT,
                error_message TEXT,
                logs TEXT,
                metrics TEXT,
```

```
            PRIMARY KEY (task_id, pipeline_run_id)
        )
    """)

    cursor.execute("""
        CREATE TABLE IF NOT EXISTS state_transitions (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            task_id TEXT,
            pipeline_run_id TEXT,
            from_state TEXT,
            to_state TEXT,
            event TEXT,
            timestamp TEXT,
            error_message TEXT
        )
    """)

    conn.commit()
    conn.close()
```

Core Logic Skeleton

Pipeline Execution Coordinator:

PYTHON

```python
"""
Main pipeline execution coordination logic.
Students implement the core orchestration algorithms here.
"""
from typing import Dict, List, Set, Optional
from datetime import datetime
import asyncio

class PipelineExecutor:
    """
    Coordinates execution of complete pipeline runs including dependency resolution,
    task scheduling, and failure handling.
    """

    def __init__(self, message_broker, state_manager, dag_engine):
        self.message_broker = message_broker
        self.state_manager = state_manager
        self.dag_engine = dag_engine

    async def execute_pipeline(self, pipeline_id: str, run_id: str, parameters: Dict[str, Any]) -> bool:
        """
        Execute complete pipeline run with dependency management and monitoring.
        Returns True if pipeline completed successfully, False otherwise.
        """
        # TODO 1: Retrieve pipeline definition using get_pipeline(pipeline_id)
        # TODO 2: Validate pipeline definition using validate_pipeline(pipeline)
        # TODO 3: Create execution plan using create_execution_plan(pipeline, task_durations)
        # TODO 4: Initialize all task executions in PENDING state
        # TODO 5: Start dependency resolution loop for task scheduling
        # TODO 6: Monitor task completion events and update dependency tracking
        # TODO 7: Handle task failures according to pipeline failure policy
        # TODO 8: Determine pipeline completion status when all tasks finish
        # TODO 9: Publish pipeline completion event with final metrics
```

```
        # TODO 10: Perform cleanup of temporary resources and state


    def _schedule_eligible_tasks(self, run_id: str, execution_plan) -> List[str]:
        """
        Schedule tasks that have all dependencies satisfied.
        Returns list of scheduled task IDs.
        """
        # TODO 1: Query current task states for this pipeline run
        # TODO 2: Find tasks in WAITING state (dependencies met but not yet scheduled)
        # TODO 3: Check resource availability for each eligible task
        # TODO 4: Assign tasks to available executors using load balancing
        # TODO 5: Update task states to QUEUED and publish scheduling events
        # TODO 6: Return list of successfully scheduled task IDs


    def _update_dependencies(self, completed_task_id: str, run_id: str) -> Set[str]:
        """
        Update dependency tracking when a task completes.
        Returns set of task IDs that became eligible for execution.
        """
        # TODO 1: Retrieve execution plan dependency matrix for this run
        # TODO 2: Find all tasks that depend on the completed task
        # TODO 3: Check if all dependencies are now satisfied for each dependent task
        # TODO 4: Transition eligible tasks from PENDING to WAITING state
        # TODO 5: Return set of task IDs that became eligible


    def _handle_task_failure(self, failed_task_id: str, run_id: str, error_message: str) -> bool:
        """
        Handle task failure according to pipeline failure policy.
        Returns True if pipeline should continue, False if it should abort.
        """
        # TODO 1: Check task retry policy and current attempt count
        # TODO 2: If retries available, schedule retry with exponential backoff
        # TODO 3: If no retries left, determine impact on downstream tasks
```

```
        # TODO 4: Apply pipeline failure policy (fail-fast, continue-on-error, etc.)

        # TODO 5: Cancel or skip dependent tasks based on failure policy

        # TODO 6: Return whether pipeline execution should continue
```

Lineage Collection Implementation:

PYTHON

```
"""
Automatic lineage collection during pipeline execution.
Students implement lineage capture and analysis logic here.
"""

class LineageCollector:
    """
    Captures data lineage information during pipeline execution.
    Supports both automatic collection and manual annotation.
    """

    def __init__(self, storage_backend):
        self.storage = storage_backend

    def collect_sql_lineage(self, sql_query: str, input_tables: List[str], output_table: str,
                            pipeline_run_id: str, task_id: str) -> bool:
        """
        Extract lineage from SQL transformations using query analysis.
        Returns True if lineage was successfully captured.
        """
        # TODO 1: Parse SQL query using SQL AST parser to extract structure
        # TODO 2: Identify all referenced tables and columns in FROM/JOIN clauses
        # TODO 3: Extract column expressions from SELECT clause with transformations
        # TODO 4: Map input columns to output columns through SELECT expressions
        # TODO 5: Create AssetReference objects for input and output tables
        # TODO 6: Build ColumnLineage mappings for each transformed column
        # TODO 7: Store complete lineage record with transformation logic
        # TODO 8: Publish lineage event for downstream consumers

    def collect_udf_lineage(self, function_name: str, input_data_schema: dict,
                            output_data_schema: dict, pipeline_run_id: str, task_id: str) -> bool:
        """
        Capture lineage for Python UDF transformations through runtime inspection.
```

```
        Returns True if lineage was successfully captured.
        """
        # TODO 1: Retrieve UDF metadata and manual lineage annotations
        # TODO 2: Compare input and output schemas to infer column mappings
        # TODO 3: Sample input and output data to detect transformation patterns
        # TODO 4: Extract business logic annotations from function docstrings
        # TODO 5: Build lineage record with available transformation information
        # TODO 6: Mark uncertain lineage relationships for manual review
        # TODO 7: Store lineage record and publish collection event

    def get_data_lineage(self, asset_name: str, direction: str = "both", max_depth: int = 10) -> dict:
        """
        Query lineage relationships for specified data asset.
        Direction can be 'upstream', 'downstream', or 'both'.
        Returns lineage graph with transformation details.
        """
        # TODO 1: Validate asset name and query parameters
        # TODO 2: Build graph traversal query based on direction and depth
        # TODO 3: Execute lineage query against storage backend
        # TODO 4: Construct lineage graph with nodes (assets) and edges (transformations)
        # TODO 5: Include transformation logic and schema evolution details
        # TODO 6: Apply access controls to filter visible lineage information
        # TODO 7: Return structured lineage graph for visualization/analysis
```

Milestone Checkpoints

Checkpoint 1 - Component Communication (End of Milestone 1):

- Run: `python -m pytest tests/integration/test_messaging.py -v`
- Expected: All message publishing and subscription tests pass
- Manual verification: Start message broker, publish test event, verify subscriber receives it
- Signs of issues: Message delivery failures, serialization errors, subscription timeouts

Checkpoint 2 - Pipeline Execution Flow (End of Milestone 2):

- Run: `python scripts/test_pipeline_execution.py sample_pipeline.yaml`
- Expected: Complete pipeline execution with task dependency resolution
- Manual verification: Monitor task state transitions through dashboard/logs
- Signs of issues: Dependency deadlocks, resource allocation failures, state inconsistencies

Checkpoint 3 - Lineage Tracking (End of Milestone 3):

- Run: `python scripts/test_lineage_collection.py`
- Expected: Automatic lineage capture for SQL and Python transformations
- Manual verification: Query lineage API to verify forward/backward relationships
- Signs of issues: Missing lineage records, incorrect column mappings, SQL parsing failures

Checkpoint 4 - Complete Integration (End of Milestone 4):

- Run: `python scripts/run_full_pipeline_test.py`
- Expected: End-to-end pipeline with monitoring, alerting, and lineage
- Manual verification: Complete pipeline run with full observability
- Signs of issues: Missing metrics, alert failures, incomplete lineage graphs

Error Handling and Edge Cases

> **Milestone(s):** All milestones - comprehensive error handling affects pipeline definition validation (Milestone 1), data processing resilience (Milestones 2-3), and orchestration reliability (Milestone 4). This section establishes the production-grade error handling patterns required across all system components.

Mental Model: Hospital Emergency Response System

Think of our error handling system like a hospital's emergency response infrastructure. When patients arrive, they don't just randomly hope for the best - there's a systematic approach to handling different types of medical emergencies. **Triage** classifies problems by severity (minor cut vs. heart attack), **treatment protocols** define standardized responses to common conditions (broken bone → X-ray → cast → recovery), and **escalation procedures** know when to call specialists or transfer to higher-level care facilities.

Similarly, our ETL system must systematically classify failures, apply appropriate treatment protocols (retry strategies), and escalate to human operators when automated recovery fails. Just as hospitals have backup generators and redundant life support systems, our pipeline infrastructure requires multiple layers of fault tolerance to ensure critical data flows never completely fail.

The key insight is that **not all failures are created equal**. A temporary network blip should trigger automatic retry with exponential backoff, while data corruption requires immediate human attention and potentially rolling back processed data. Like medical triage, proper classification at the moment of failure determines whether the patient (pipeline) recovers quickly or suffers permanent damage.

Common Failure Modes

Production ETL systems face numerous failure scenarios that can disrupt data processing, corrupt results, or leave pipelines in inconsistent states. Understanding these failure modes enables us to design appropriate detection, classification, and recovery mechanisms.

Network and Connectivity Failures

Network-related failures represent the most common category of transient errors in distributed ETL systems. These failures manifest as connection timeouts, DNS resolution failures, SSL handshake errors, and intermittent packet loss that can cause partial data corruption during transmission.

Connection timeouts occur when source systems become temporarily overloaded or network congestion prevents timely responses. Database connections may time out during long-running extraction queries, while API endpoints may become unresponsive during peak usage periods. The challenge lies in distinguishing between temporary overload (retry appropriate) and permanent service degradation (escalation required).

DNS resolution failures can prevent pipeline tasks from reaching their target systems entirely. These failures often indicate infrastructure-level problems that affect multiple pipelines simultaneously. Unlike connection timeouts, DNS failures typically require

immediate escalation since they suggest broader network infrastructure issues.

SSL certificate issues create authentication failures that prevent secure connections to external systems. Certificate expiration, hostname mismatches, and certificate authority changes can break previously working connections. These failures require immediate attention since they often indicate security-related configuration drift.

Partial network failures represent the most insidious category, where connections succeed initially but encounter intermittent packet loss or bandwidth restrictions. Large data transfers may partially complete before failing, leaving destination systems in inconsistent states that require careful cleanup.

Failure Type	Symptoms	Detection Method	Recovery Strategy
Connection Timeout	HTTP 408, socket timeout exceptions	Connection monitoring, response time alerts	Exponential backoff retry, connection pool cycling
DNS Resolution Failure	Name resolution errors, DNS lookup timeouts	DNS health checks, resolution time monitoring	Immediate escalation, fallback DNS servers
SSL Certificate Issues	Certificate validation errors, handshake failures	Certificate expiration monitoring, handshake alerts	Immediate escalation, certificate renewal workflow
Partial Network Failure	Incomplete transfers, checksum mismatches	Transfer verification, data integrity checks	Resume from checkpoint, full transfer retry

Data Quality and Corruption Issues

Data quality problems represent critical failures that can propagate incorrect information throughout downstream systems. Unlike network failures, data corruption issues require immediate human intervention and potentially complex rollback procedures.

Schema evolution conflicts occur when source systems change their data structures without coordinating with ETL pipelines. New required fields, removed columns, or data type changes can cause extraction or transformation tasks to fail catastrophically. The fundamental challenge is distinguishing between temporary schema access issues and permanent structural changes.

Data format corruption manifests when source systems produce malformed records that don't conform to expected schemas. JSON parsing errors, CSV files with inconsistent column counts, and binary data corruption represent common examples. These failures require sophisticated validation to prevent corrupt records from contaminating destination systems.

Referential integrity violations occur when extracted data references entities that don't exist in destination systems. Foreign key constraint violations, orphaned records, and circular references can prevent bulk loading operations from completing successfully.

Data volume anomalies indicate potential upstream system failures or data generation issues. Sudden spikes or drops in record counts, unusually large field values, or completely empty extractions suggest problems that require immediate investigation.

Data Issue Type	Detection Method	Impact Assessment	Recovery Action
Schema Evolution	Field validation, type checking failures	High - affects all downstream consumers	Stop pipeline, schema compatibility analysis
Format Corruption	Parse errors, validation failures	Medium - affects individual records	Quarantine invalid records, continue processing
Referential Integrity	Foreign key violations, constraint errors	High - data consistency compromised	Rollback transaction, dependency resolution
Volume Anomalies	Record count monitoring, statistical analysis	Variable - depends on business context	Alert operators, apply business rule validation

Resource Exhaustion and Performance Degradation

Resource exhaustion represents a class of failures where system resources become insufficient to support normal pipeline operations. These failures often develop gradually and can affect multiple pipelines simultaneously.

Memory exhaustion occurs when transformation operations attempt to load datasets larger than available RAM. Aggregation operations, large joins, and bulk data processing can trigger out-of-memory errors that terminate pipeline tasks abruptly. The challenge lies in detecting memory pressure early enough to implement mitigation strategies.

Disk space exhaustion affects both temporary processing storage and permanent result storage. Staging tables, intermediate transformation results, and log files can consume available disk space, preventing pipeline completion. These failures require both immediate cleanup and longer-term capacity planning.

Connection pool exhaustion happens when concurrent pipeline tasks exceed the maximum number of available database connections. This creates resource starvation where new tasks cannot acquire necessary connections to proceed.

CPU and I/O bottlenecks manifest as severely degraded performance that can cause pipeline tasks to exceed their timeout limits. While not technically failures, extreme performance degradation often leads to timeout-based task cancellation.

> **Critical Insight**: Resource exhaustion failures often cascade across multiple pipelines. A single memory-intensive transformation can consume available RAM, causing unrelated pipelines to fail due to resource starvation. Detection and mitigation must consider system-wide resource allocation, not just individual pipeline requirements.

Resource Type	Exhaustion Symptoms	Monitoring Approach	Mitigation Strategy
Memory	OutOfMemoryError, swap thrashing	Memory usage monitoring, GC pressure alerts	Streaming processing, data partitioning
Disk Space	Write failures, temp file creation errors	Disk usage monitoring, growth rate analysis	Cleanup temporary files, partition pruning
Connection Pool	Connection acquisition timeouts	Active connection monitoring, pool saturation alerts	Connection pool sizing, connection recycling
CPU/I/O	Extreme response times, timeout failures	Performance monitoring, resource utilization tracking	Task parallelization limits, resource quotas

Retry and Backoff Strategies

Effective retry strategies distinguish between transient failures that will likely resolve themselves and permanent failures that require human intervention. The goal is to achieve maximum reliability without overwhelming already-stressed systems with aggressive retry attempts.

Exponential Backoff Implementation

Exponential backoff provides a mathematically sound approach to spacing retry attempts that reduces load on failing systems while maintaining reasonable recovery times for transient issues. The strategy involves doubling the delay between successive retry attempts, with optional jitter to prevent thundering herd effects.

The basic exponential backoff formula calculates delay as `initial_delay * (2^attempt_count)`, with maximum delay caps to prevent indefinitely long wait times. However, naive implementation can create synchronized retry storms when multiple pipeline tasks fail simultaneously and begin retrying on identical schedules.

Jitter injection addresses synchronized retry problems by introducing randomness to retry timing. Full jitter randomizes the entire delay window, while decorrelated jitter uses the previous delay as input to random number generation. The choice between jitter strategies depends on the expected correlation between failure events and system recovery characteristics.

Retry budget management prevents retry attempts from consuming excessive execution time relative to useful work. Each task maintains a retry budget that limits total retry time or attempt counts, ensuring that persistently failing tasks don't block pipeline completion indefinitely.

> **Decision: Exponential Backoff with Decorrelated Jitter**
>
> - **Context**: Multiple pipeline tasks often fail simultaneously due to shared dependencies (database, network), creating retry storms that can overwhelm recovering systems
> - **Options Considered**: Fixed delay, exponential backoff, exponential backoff with full jitter, decorrelated jitter
> - **Decision**: Exponential backoff with decorrelated jitter and maximum delay caps
> - **Rationale**: Decorrelated jitter spreads retry attempts across time while maintaining reasonable mathematical properties. Maximum delay caps prevent indefinitely long waits that could block pipeline completion.
> - **Consequences**: Eliminates retry storms and reduces system load during recovery, at the cost of slightly more complex retry timing calculation

Retry Strategy	Formula	Pros	Cons
Fixed Delay	`delay = constant`	Simple, predictable timing	Can create retry storms, ignores system recovery
Exponential Backoff	`delay = initial * (2^attempt)`	Reduces load on failing systems	Can create synchronized retry storms
Full Jitter	`delay = random(0, exponential_delay)`	Eliminates synchronization	Can retry too quickly or too slowly
Decorrelated Jitter	`delay = random(initial, previous_delay * 3)`	Balances timing with load spreading	More complex calculation

The `RetryPolicy` configuration allows per-task customization of retry behavior based on the expected failure characteristics of different operation types:

Field	Type	Description	Default Value
max_attempts	int	Maximum number of retry attempts before giving up	3
backoff_seconds	int	Initial delay in seconds before first retry	1
exponential_backoff	bool	Whether to use exponential backoff or fixed delay	true
retry_on_error_types	List[str]	List of error types that should trigger retries	["NetworkError", "TimeoutError"]

Circuit Breaker Pattern

Circuit breakers protect downstream systems from cascading failures by temporarily suspending requests to systems that are experiencing problems. Like electrical circuit breakers, they "trip" when failure rates exceed acceptable thresholds and automatically "reset" when systems recover.

The circuit breaker maintains three states: **Closed** (normal operation), **Open** (failures detected, requests blocked), and **Half-Open** (testing recovery). State transitions depend on failure rate monitoring and recovery detection logic.

Failure rate calculation requires sliding window analysis to distinguish between temporary failure spikes and sustained system degradation. Count-based windows track failures over a fixed number of recent requests, while time-based windows monitor failure rates over rolling time periods.

Recovery detection in the Half-Open state allows a limited number of test requests to determine if the downstream system has recovered. Successful test requests transition the circuit back to Closed state, while continued failures return to Open state with potentially extended timeout periods.

Circuit State	Request Handling	Transition Trigger	Typical Duration
Closed	Forward all requests	Failure rate exceeds threshold	N/A - normal operation
Open	Reject requests immediately	Timeout period expires	60-300 seconds
Half-Open	Allow limited test requests	Successful/failed test requests	5-30 seconds

Dead Letter Queue Management

Dead letter queues provide a mechanism for handling messages or tasks that cannot be processed successfully after exhausting all retry attempts. Rather than losing failed operations entirely, dead letter queues preserve them for later analysis and potential reprocessing.

Message preservation ensures that failed operations retain all necessary context for debugging and recovery. This includes original task parameters, error messages from all retry attempts, execution timing information, and environmental context like pipeline run identifiers.

Retry exhaustion criteria determine when tasks should be moved to dead letter queues. Simple count-based criteria move tasks after a fixed number of attempts, while time-based criteria consider total retry duration. Sophisticated approaches consider error type classification, with permanent errors moving immediately to dead letter queues.

Dead letter processing enables batch reprocessing of failed operations after resolving underlying issues. Manual reprocessing allows operators to modify task parameters or execution context, while automatic reprocessing can retry operations when system health metrics indicate recovery.

> **Architecture Decision: Separate Dead Letter Queues by Failure Type**
>
> - **Context**: Different failure types require different reprocessing approaches and have different urgency levels for human intervention
> - **Options Considered**: Single dead letter queue, separate queues by pipeline, separate queues by failure type
> - **Decision**: Separate dead letter queues categorized by failure type (transient, data quality, system error)
> - **Rationale**: Enables targeted reprocessing strategies and appropriate alert prioritization. Data quality issues need immediate attention while transient failures can be batch processed.
> - **Consequences**: Requires failure classification logic and multiple queue management, but provides better operational control and reduced alert noise.

Dead Letter Queue Type	Failure Categories	Reprocessing Strategy	Alert Priority
Transient Failures	Network timeouts, temporary resource exhaustion	Automatic batch retry during off-peak hours	Low
Data Quality Issues	Schema validation, referential integrity	Manual review and correction required	High
System Errors	Configuration issues, permission problems	Requires system administrator intervention	Medium
Permanent Failures	Invalid configurations, missing resources	Manual task modification or removal	Low

Partial Failure Recovery

Partial failures represent some of the most challenging scenarios in distributed ETL systems. Unlike complete failures where the entire operation fails cleanly, partial failures leave systems in intermediate states that require careful analysis and recovery procedures.

Checkpointing and State Persistence

Checkpointing enables pipelines to resume processing from intermediate points rather than restarting from the beginning after failures. Effective checkpointing requires identifying appropriate checkpoint boundaries, persisting sufficient state information, and implementing recovery logic that can resume from any checkpoint.

Checkpoint boundary selection depends on the natural granularity of data processing operations. Bulk loading operations might checkpoint after each batch of records, while transformation operations might checkpoint after processing each input partition. The goal is to balance checkpoint frequency (more frequent checkpoints reduce rework) against checkpoint overhead (state persistence costs).

State persistence requirements extend beyond simple progress tracking to include all information necessary to resume processing. This includes input data positions (file offsets, database cursors), intermediate calculation results, configuration parameters, and dependency state. The challenge lies in ensuring checkpoint state remains consistent even if the checkpointing process itself is interrupted.

Recovery logic complexity grows significantly with checkpoint granularity. Coarse-grained checkpoints require simpler recovery logic but result in more rework after failures. Fine-grained checkpoints minimize rework but require sophisticated logic to handle partially completed operations and state inconsistencies.

> **Critical Design Principle**: Checkpoints must be **atomic** and **consistent**. A checkpoint is only valid if all related state can be persisted atomically. Partial checkpoints can leave the system in an unrecoverable state that's worse than no checkpoint at all.

The `TaskExecution` model supports checkpointing through the metrics field, which can store arbitrary checkpoint data:

Checkpoint Information	Storage Location	Recovery Usage
Input data position	metrics["input_position"]	Resume reading from exact location
Processed record count	metrics["records_processed"]	Validate recovery completeness
Intermediate results	metrics["intermediate_state"]	Avoid recalculating expensive operations
Error recovery state	metrics["error_context"]	Understand failure context for recovery

Transaction Management and Rollback

Transaction management ensures that data operations either complete entirely or leave no traces in destination systems. ETL operations often involve multiple systems (source extraction, staging tables, destination loading) that must be coordinated to maintain consistency.

Two-phase commit protocols can coordinate transactions across multiple systems, but introduce significant complexity and performance overhead. The first phase involves sending prepare messages to all participating systems, while the second phase sends commit or abort decisions based on unanimous agreement.

Compensation-based transactions provide an alternative approach where each operation includes a corresponding compensation action that undoes its effects. Rather than preventing inconsistent states, compensation transactions restore consistency after detecting failures.

Saga pattern implementation breaks long-running transactions into sequences of smaller transactions, each with its own compensation action. If any step fails, the saga executes compensation actions for all completed steps in reverse order.

Transaction Approach	Coordination Method	Consistency Guarantee	Performance Impact
Two-Phase Commit	Distributed consensus	Strong consistency	High latency, blocking
Compensation Transactions	Reverse operation execution	Eventual consistency	Lower latency, non-blocking
Saga Pattern	Sequential compensation	Eventual consistency	Moderate latency, complex logic

Data Cleanup and Consistency Repair

Data cleanup operations remove incomplete or corrupted data that results from partial failures. The challenge lies in identifying which data was affected by failures and determining safe cleanup boundaries that don't accidentally remove valid data.

Staging table cleanup involves removing partially loaded data from staging areas after extraction or transformation failures. Simple approaches truncate entire staging tables, while sophisticated approaches identify and remove only records from failed operations.

Referential integrity repair addresses situations where partial failures create orphaned records or broken foreign key relationships. Repair operations might involve removing orphaned records, restoring missing parent records, or updating references to point to valid entities.

Idempotent operation design enables safe retry of cleanup operations without risking data corruption. Cleanup operations should produce identical results regardless of how many times they're executed, even if the system state changes between executions.

⚠️ Pitfall: Cleanup Operations Without Transaction Boundaries

Many developers implement cleanup operations as simple DELETE statements without considering transaction boundaries. If the cleanup operation itself fails partway through, it can leave the system in a state that's more inconsistent than before cleanup began.

Why it's wrong: Partial cleanup can remove some corrupted data while leaving other corrupted data in place, making it harder to identify the scope of data quality issues.

How to fix: Wrap cleanup operations in transactions and implement cleanup checkpointing for operations that affect large data volumes. Consider using staging tables for cleanup operations so that incomplete cleanup doesn't affect production data.

Cleanup Operation Type	Scope Identification	Safety Mechanism	Recovery Approach
Staging Table Cleanup	Pipeline run ID, timestamp ranges	Transaction boundaries	Truncate and reload
Partial Load Cleanup	Batch identifiers, watermark ranges	Backup before cleanup	Restore from backup
Referential Integrity Repair	Foreign key violation detection	Constraint validation	Cascade repair operations
Orphaned Record Cleanup	Parent-child relationship analysis	Soft delete before hard delete	Restoration from soft delete

Implementation Guidance

Understanding error handling patterns requires implementing robust failure detection, classification, and recovery mechanisms. The following guidance provides concrete implementation approaches for production-grade error handling.

Technology Recommendations

Component	Simple Option	Advanced Option
Retry Logic	Simple exponential backoff with random jitter	Redis-backed circuit breaker with sliding window
Dead Letter Queue	Database table with failed task records	Message broker (RabbitMQ/Apache Kafka) with topic-based routing
Circuit Breaker	In-memory failure tracking with timeout-based reset	Distributed circuit breaker with shared state
Checkpointing	JSON files with task state snapshots	Database transactions with write-ahead logging
Monitoring	Structured logging with failure categorization	Metrics collection (Prometheus) with alerting rules

Recommended File Structure

```
etl-system/
  core/
    error_handling/
      __init__.py
      retry_policy.py           ← RetryPolicy and exponential backoff
      circuit_breaker.py        ← Circuit breaker implementation
      dead_letter_queue.py      ← Dead letter queue management
      checkpoint_manager.py     ← Checkpointing and state persistence
      error_classifier.py       ← Failure type classification
      recovery_engine.py        ← Automated recovery procedures
  tests/
    error_handling/
      test_retry_policy.py
      test_circuit_breaker.py
      test_checkpoint_manager.py
  monitoring/
    error_metrics.py          ← Error metrics collection
    alert_manager.py          ← Alert generation and suppression
```

Infrastructure Starter Code

RetryPolicy with Exponential Backoff:

```python
import random

import time

from typing import List, Optional

from dataclasses import dataclass

from enum import Enum


class ErrorType(Enum):

    NETWORK_ERROR = "NetworkError"

    TIMEOUT_ERROR = "TimeoutError"

    DATA_QUALITY_ERROR = "DataQualityError"

    RESOURCE_EXHAUSTION = "ResourceExhaustion"

    PERMISSION_ERROR = "PermissionError"


@dataclass

class RetryPolicy:

    max_attempts: int = 3

    backoff_seconds: int = 1

    exponential_backoff: bool = True

    retry_on_error_types: List[str] = None

    max_delay_seconds: int = 300

    jitter_type: str = "decorrelated"



    def __post_init__(self):

        if self.retry_on_error_types is None:

            self.retry_on_error_types = [

                ErrorType.NETWORK_ERROR.value,

                ErrorType.TIMEOUT_ERROR.value,

                ErrorType.RESOURCE_EXHAUSTION.value

            ]

class RetryManager:

    def __init__(self, retry_policy: RetryPolicy):

        self.policy = retry_policy
```

```
    def should_retry(self, error_type: str, attempt_count: int) -> bool:
        """Determine if operation should be retried based on policy and error type."""
        if attempt_count >= self.policy.max_attempts:
            return False
        return error_type in self.policy.retry_on_error_types

    def calculate_delay(self, attempt_count: int, previous_delay: Optional[int] = None) -> int:
        """Calculate delay before next retry attempt."""
        if not self.policy.exponential_backoff:
            return self.policy.backoff_seconds

        if self.policy.jitter_type == "decorrelated" and previous_delay:
            # Decorrelated jitter: random(initial_delay, previous_delay * 3)
            min_delay = self.policy.backoff_seconds
            max_delay = min(previous_delay * 3, self.policy.max_delay_seconds)
            return random.randint(min_delay, max_delay)
        else:
            # Standard exponential backoff with full jitter
            base_delay = self.policy.backoff_seconds * (2 ** attempt_count)
            max_delay = min(base_delay, self.policy.max_delay_seconds)
            return random.randint(0, max_delay)

    def execute_with_retry(self, operation, error_classifier, *args, **kwargs):
        """Execute operation with retry logic based on policy."""
        last_exception = None
        previous_delay = None

        for attempt in range(self.policy.max_attempts):
            try:
                return operation(*args, **kwargs)
            except Exception as e:
                error_type = error_classifier.classify_error(e)
                last_exception = e
```

```
            if not self.should_retry(error_type, attempt + 1):
                break

            delay = self.calculate_delay(attempt, previous_delay)
            previous_delay = delay
            time.sleep(delay)

        raise last_exception
```

Circuit Breaker Implementation:

PYTHON

```python
import time

import threading

from enum import Enum

from dataclasses import dataclass

from typing import Callable, Any


class CircuitState(Enum):

    CLOSED = "closed"

    OPEN = "open"

    HALF_OPEN = "half_open"


@dataclass

class CircuitBreakerConfig:

    failure_threshold: int = 5

    success_threshold: int = 3

    timeout_seconds: int = 60

    window_size: int = 100


class CircuitBreaker:

    def __init__(self, config: CircuitBreakerConfig):

        self.config = config

        self.state = CircuitState.CLOSED

        self.failure_count = 0

        self.success_count = 0

        self.last_failure_time = 0

        self.request_count = 0

        self.lock = threading.Lock()



    def call(self, operation: Callable, *args, **kwargs) -> Any:

        """Execute operation through circuit breaker."""

        with self.lock:

            if self.state == CircuitState.OPEN:

                if time.time() - self.last_failure_time < self.config.timeout_seconds:

                    raise CircuitBreakerOpenError("Circuit breaker is open")
```

```
            else:
                self.state = CircuitState.HALF_OPEN
                self.success_count = 0

    try:
        result = operation(*args, **kwargs)
        self._record_success()
        return result
    except Exception as e:
        self._record_failure()
        raise

def _record_success(self):
    """Record successful operation and potentially close circuit."""
    with self.lock:
        if self.state == CircuitState.HALF_OPEN:
            self.success_count += 1
            if self.success_count >= self.config.success_threshold:
                self.state = CircuitState.CLOSED
                self.failure_count = 0
        elif self.state == CircuitState.CLOSED:
            self.failure_count = max(0, self.failure_count - 1)

def _record_failure(self):
    """Record failed operation and potentially open circuit."""
    with self.lock:
        self.failure_count += 1
        self.last_failure_time = time.time()

        if self.failure_count >= self.config.failure_threshold:
            self.state = CircuitState.OPEN
        elif self.state == CircuitState.HALF_OPEN:
            self.state = CircuitState.OPEN
```

```
class CircuitBreakerOpenError(Exception):
    """Raised when circuit breaker is open and blocking requests."""
    pass
```

Core Logic Skeleton Code

Error Classification System:

```python
from typing import Dict, Type, List

from enum import Enum

import re


class ErrorClassifier:

    def __init__(self):

        self.classification_rules = self._build_classification_rules()



    def classify_error(self, exception: Exception) -> str:

        """Classify exception into error type category for retry decision."""

        # TODO 1: Check exception type against known error type mappings

        # TODO 2: Examine exception message for pattern-based classification

        # TODO 3: Consider exception context (network, database, file system)

        # TODO 4: Return appropriate ErrorType enum value

        # TODO 5: Default to non-retryable error for unknown exception types

        pass



    def is_retryable(self, error_type: str) -> bool:

        """Determine if error type should trigger retry attempts."""

        # TODO 1: Check error_type against list of retryable categories

        # TODO 2: Consider system state (circuit breaker status, resource availability)

        # TODO 3: Return boolean indicating retry appropriateness

        pass



    def _build_classification_rules(self) -> Dict[str, List[str]]:

        """Build mapping of error patterns to classification categories."""

        # TODO 1: Define regex patterns for network-related errors

        # TODO 2: Define patterns for database connection and timeout errors

        # TODO 3: Define patterns for resource exhaustion (memory, disk, connections)

        # TODO 4: Define patterns for data quality and validation errors

        # TODO 5: Return comprehensive pattern mapping dictionary

        pass
```

Checkpoint Manager:

PYTHON

```python
from typing import Dict, Any, Optional
import json
import os
from datetime import datetime

class CheckpointManager:
    def __init__(self, checkpoint_dir: str):
        self.checkpoint_dir = checkpoint_dir
        os.makedirs(checkpoint_dir, exist_ok=True)

    def save_checkpoint(self, task_id: str, pipeline_run_id: str,
                        checkpoint_data: Dict[str, Any]) -> bool:
        """Save task checkpoint data for recovery purposes."""
        # TODO 1: Create checkpoint record with task_id, run_id, and timestamp
        # TODO 2: Serialize checkpoint_data to JSON with error handling
        # TODO 3: Write to temporary file first, then atomic rename
        # TODO 4: Verify checkpoint file integrity after writing
        # TODO 5: Return success/failure status
        # Hint: Use os.rename() for atomic file operations
        pass

    def load_checkpoint(self, task_id: str, pipeline_run_id: str) -> Optional[Dict[str, Any]]:
        """Load most recent checkpoint data for task recovery."""
        # TODO 1: Construct checkpoint filename from task_id and run_id
        # TODO 2: Check if checkpoint file exists and is readable
        # TODO 3: Load and deserialize JSON data with error handling
        # TODO 4: Validate checkpoint data structure and completeness
        # TODO 5: Return checkpoint data or None if not found/invalid
        pass

    def cleanup_old_checkpoints(self, retention_hours: int = 24) -> int:
        """Remove checkpoint files older than retention period."""
        # TODO 1: Scan checkpoint directory for all checkpoint files
```

```
        # TODO 2: Parse timestamps from filenames or file modification times
        # TODO 3: Identify files older than retention_hours
        # TODO 4: Remove old files with proper error handling
        # TODO 5: Return count of files removed
        pass
```

Dead Letter Queue Manager:

PYTHON

```python
from typing import List, Dict, Any, Optional
from dataclasses import dataclass
from datetime import datetime
import json

@dataclass
class DeadLetterMessage:
    task_id: str
    pipeline_run_id: str
    error_type: str
    error_message: str
    retry_count: int
    original_payload: Dict[str, Any]
    failed_at: datetime


class DeadLetterQueueManager:
    def __init__(self, storage_backend: str = "database"):
        self.storage_backend = storage_backend
        self._initialize_storage()


    def send_to_dead_letter_queue(self, message: DeadLetterMessage) -> bool:
        """Send failed task to appropriate dead letter queue."""
        # TODO 1: Determine dead letter queue category based on error_type
        # TODO 2: Serialize message data for storage
        # TODO 3: Store message in appropriate dead letter queue
        # TODO 4: Update dead letter queue metrics and alerts
        # TODO 5: Return success status
        pass


    def retrieve_messages(self, queue_type: str, limit: int = 100) -> List[DeadLetterMessage]:
        """Retrieve messages from dead letter queue for reprocessing."""
        # TODO 1: Query storage backend for messages of specified queue_type
        # TODO 2: Deserialize message data back to DeadLetterMessage objects
```

```
        # TODO 3: Apply limit to prevent memory exhaustion
        # TODO 4: Mark retrieved messages as "in_progress" to prevent duplicate processing
        # TODO 5: Return list of messages ready for reprocessing
        pass

    def requeue_message(self, message_id: str, new_retry_policy: Optional[Dict] = None) -> bool:
        """Move message from dead letter queue back to normal processing."""
        # TODO 1: Retrieve original message data from dead letter queue
        # TODO 2: Apply new retry policy if provided
        # TODO 3: Create new task execution record for retry
        # TODO 4: Remove message from dead letter queue
        # TODO 5: Submit task to normal execution queue
        pass
```

Milestone Checkpoints

After implementing error handling components, verify the following behaviors:

Checkpoint 1 - Retry Logic Verification:

```bash
# Run retry policy tests
python -m pytest tests/error_handling/test_retry_policy.py -v

# Expected output should show:
# - Exponential backoff calculations with jitter
# - Proper error type classification and retry decisions
# - Maximum retry limits respected
# - Circuit breaker state transitions
```

Checkpoint 2 - Dead Letter Queue Processing:

```bash
# Simulate pipeline failure and dead letter queue processing
python scripts/simulate_pipeline_failure.py --task-id test-task --error-type NetworkError

# Expected behavior:
# - Failed task appears in dead letter queue after max retries
# - Error classification matches expected category
# - Requeue functionality moves task back to processing
```

Checkpoint 3 - Checkpoint Recovery:

```bash
# Test checkpoint save/restore functionality
python scripts/test_checkpoint_recovery.py --pipeline-id test-pipeline --simulate-failure

# Expected behavior:
# - Pipeline creates checkpoints at regular intervals
# - After simulated failure, pipeline resumes from last checkpoint
# - No duplicate processing of checkpointed data
```

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Tasks retry indefinitely	Error type classified as retryable but actually permanent	Check error classification logs and retry attempt patterns	Update error classification rules to identify permanent errors
Circuit breaker never opens	Failure threshold too high or failures not being counted	Monitor failure count metrics and circuit breaker state transitions	Lower failure threshold or fix failure detection logic
Checkpoints cause performance issues	Checkpoint frequency too high or checkpoint data too large	Profile checkpoint save times and storage usage	Reduce checkpoint frequency or optimize checkpoint data size
Dead letter queue grows unbounded	No dead letter queue processing or requeue logic	Monitor dead letter queue size and processing rate	Implement automated dead letter queue processing
Recovery produces duplicate data	Checkpoint boundaries don't align with transaction boundaries	Check for data duplication after recovery operations	Align checkpoints with atomic operation boundaries

Testing Strategy

> **Milestone(s):** All milestones - comprehensive testing approach validates pipeline definition (Milestone 1), data processing reliability (Milestones 2-3), and orchestration correctness (Milestone 4)

Mental Model: Quality Assurance Factory

Think of our testing strategy as a multi-stage quality assurance factory for complex machinery. Just as a car manufacturer tests individual components (engines, transmissions, brakes) in isolation before assembling them into complete vehicles, and then tests those vehicles under various road conditions, our ETL system requires testing at multiple levels. We test individual components like cycle detection algorithms and data connectors in isolation (unit testing), then test how they work together with real data sources (integration testing), and finally verify that each major assembly milestone produces working functionality (milestone checkpoints). Each testing stage catches different types of defects - unit tests catch logic errors, integration tests catch interface mismatches, and milestone checkpoints catch system-level failures.

The key insight is that ETL systems have unique testing challenges compared to typical web applications. They process large volumes of data, interact with external systems that may be unreliable, and have complex dependency graphs that can fail in subtle ways. Our testing strategy must address data quality issues, handle non-deterministic external dependencies, and validate both correctness and performance characteristics under realistic load conditions.

Unit Testing Approach

Component Isolation Strategy

Unit testing in an ETL system requires careful isolation of components that normally interact with external systems, maintain state across operations, and process large data volumes. The primary challenge is creating reliable, fast tests that validate core logic without depending on databases, file systems, or network services.

Our unit testing approach focuses on testing individual algorithms, data transformations, and business logic in complete isolation. Each test should run in milliseconds, be deterministic regardless of execution order, and require no external dependencies. This means mocking or stubbing any interactions with databases, APIs, file systems, or message brokers.

Core Component Testing Areas

The following table outlines the major component categories and their specific testing requirements:

Component Type	Primary Test Focus	Mock Dependencies	Key Test Scenarios
DAG Engine	Cycle detection, topological sorting	None (pure algorithms)	Valid DAGs, circular dependencies, complex graphs
Connectors	Query generation, pagination logic	Database connections, HTTP clients	SQL templating, cursor handling, error responses
Transformations	Data type conversion, null handling	External data sources	Type coercion, schema validation, edge cases
State Machine	Task state transitions	Persistence layer, message broker	Valid transitions, invalid events, retry logic
Scheduler	Cron parsing, next run calculation	System clock, database	Schedule expressions, timezone handling, DST
Orchestrator	Execution order, resource allocation	Task executors, monitoring	Parallel execution, failure propagation, cleanup

Data Structure Validation Tests

Every data structure defined in our system requires comprehensive validation testing. These tests verify that validation functions correctly identify invalid data and that serialization/deserialization preserves data integrity across system boundaries.

For `PipelineDefinition` validation, we test scenarios including missing required fields, invalid cron expressions, circular task dependencies, invalid parameter types, and malformed task configurations. Each validation test should verify that appropriate error messages are generated with sufficient detail for debugging.

For `TaskExecution` state management, we test state transitions under normal conditions and edge cases including invalid state combinations, concurrent state modifications, and recovery from corrupted state. These tests use mock storage backends to avoid dependency on persistent storage systems.

Algorithm Testing with Edge Cases

The DAG processing algorithms require particularly thorough testing because they handle complex graph structures that can fail in subtle ways. The `detect_cycles_dfs` function must be tested with various graph topologies including simple cycles, complex interconnected cycles, self-loops, disconnected components, and graphs with thousands of nodes to verify performance characteristics.

The `topological_sort_kahns` algorithm requires testing with scenarios including linear chains (no parallelism), fully parallel tasks (no dependencies), diamond-shaped dependencies, and graphs with multiple valid topological orderings. We verify that the algorithm correctly identifies parallelizable task groups and handles edge cases like empty graphs and single-node graphs.

Mock Strategy for External Dependencies

External dependency mocking follows a consistent pattern across all components. Database connections are mocked using test doubles that simulate query execution, connection failures, and timeout scenarios. HTTP clients are mocked to return predefined responses, simulate network errors, and test pagination edge cases.

The following table shows our mocking approach for each external system type:

External System	Mock Implementation	Test Scenarios	Failure Simulation
Database	In-memory query executor	Query results, schema info	Connection timeout, query failure
REST API	HTTP response simulator	JSON responses, pagination	Network error, rate limiting
File System	In-memory file operations	File read/write, directory listing	Permission denied, disk full
Message Broker	Event queue simulator	Message delivery, ordering	Message loss, duplicate delivery
Time/Clock	Controllable time source	Schedule calculation, timeouts	Clock drift, timezone changes

Data Transformation Testing

Data transformation testing requires careful handling of data type edge cases, null value semantics, and schema evolution scenarios. We create test datasets that include boundary values for each data type, various null representations, and malformed data that should trigger validation errors.

Type conversion testing covers all supported `DataType` combinations, including lossy conversions that should generate warnings, invalid conversions that should fail, and precision preservation for numeric types. Each conversion test includes boundary values like maximum integers, special floating-point values (NaN, infinity), and edge cases like empty strings and whitespace-only values.

Schema validation testing uses controlled test schemas to verify that validation correctly identifies required field violations, type mismatches, constraint violations, and unknown fields. We test both strict validation (reject any violations) and lenient validation (warn but continue processing) modes.

Performance and Memory Testing

Unit tests include performance benchmarks for critical algorithms to catch performance regressions during development. The DAG algorithms are benchmarked with graphs of various sizes to verify that performance scales appropriately with graph complexity.

Memory usage testing is particularly important for data processing components that handle large datasets. We use controlled test datasets to verify that streaming operations maintain constant memory usage regardless of input size, and that batch operations properly release memory after processing.

Common Pitfalls in Unit Testing

⚠️ **Pitfall: Testing with Real External Systems** Many developers write unit tests that connect to actual databases or APIs during development. This makes tests slow, non-deterministic, and dependent on external system availability. Instead, use mock implementations that simulate the exact interface behavior without external dependencies.

⚠️ **Pitfall: Insufficient Edge Case Coverage** ETL systems handle diverse data types and formats, making edge case testing critical. Don't just test happy path scenarios - include boundary values, malformed data, and unusual but valid input combinations. Create comprehensive test data generators that cover the full range of possible inputs.

⚠️ **Pitfall: Ignoring Concurrent Access Patterns** Many ETL components will run concurrently in production. Unit tests should include scenarios with concurrent access to shared resources, using techniques like goroutines with channels or threading libraries to simulate race conditions and verify thread safety.

Integration Testing

End-to-End Pipeline Validation

Integration testing validates that our ETL components work correctly when combined with real external systems and realistic data volumes. Unlike unit tests that focus on individual component logic, integration tests verify that data flows correctly through the entire pipeline, that external system interactions handle real-world conditions, and that performance meets requirements under realistic load.

The primary challenge in integration testing is creating realistic test environments that include representative data sources, network conditions, and system load without requiring full production infrastructure. We accomplish this through containerized test environments, synthetic data generation, and careful test data management.

Test Environment Architecture

Our integration test environment uses containerized services to simulate production dependencies while maintaining test isolation and repeatability. The test environment includes database containers with realistic schemas and data volumes, mock API services that simulate third-party systems, and message broker containers for testing event-driven components.

Each test run starts with a clean environment state, loads controlled test data, executes the pipeline under test, and validates the results against expected outcomes. Test data is carefully crafted to include realistic data distributions, common data quality issues, and edge cases that occur in production systems.

The following table outlines our test environment components:

Environment Component	Implementation	Purpose	Data Volume
Source Database	PostgreSQL container	Realistic SQL extraction testing	10K-100K records
API Mock Service	HTTP server with predefined responses	API connector testing	Configurable pagination
Target Database	PostgreSQL container	Bulk loading validation	Variable based on test
Message Broker	Redis container	Event-driven testing	Message ordering validation
File Storage	Local filesystem mount	File-based extraction	Multiple file formats
Monitoring Stack	Prometheus/Grafana containers	Metrics collection testing	Full metric pipeline

Data Source Integration Testing

Database connector testing uses real database instances with controlled test schemas and data. We test against multiple database types (PostgreSQL, MySQL, SQL Server) to verify that our SQL generation and result parsing work across different database dialects and driver implementations.

Test scenarios include complex join queries, large result sets that require pagination, schema introspection for automatic mapping, and connection pooling under concurrent access. We simulate database failures including connection timeouts, query timeouts, and temporary network partitions to verify that error handling and retry logic work correctly.

API connector testing uses both real external APIs (where available and appropriate) and sophisticated mock services that simulate real API behavior. Mock services implement realistic pagination patterns, rate limiting, authentication flows, and error responses that match actual API behavior.

The following integration test scenarios validate API connector robustness:

Test Scenario	Validation Focus	Expected Behavior
Large dataset pagination	Cursor stability across pages	Complete data extraction without duplicates
Rate limit handling	Backoff and retry logic	Automatic throttling with eventual success
Authentication token refresh	Token expiration handling	Seamless authentication renewal
Partial API failures	Individual request failures	Retry failed requests without re-fetching successful pages
Network instability	Connection interruption recovery	Resume from last successful position

Transformation Pipeline Integration

Transformation integration testing validates that SQL-based transformations execute correctly against real database engines and that Python UDF execution handles realistic data volumes and types. We test transformation chains where the output of one transformation becomes the input to subsequent transformations, verifying that data types are preserved correctly across the chain.

Schema evolution testing uses versioned test schemas to verify that our transformation pipeline handles schema changes gracefully. We test scenarios including adding columns, changing column types, renaming columns, and removing columns, validating that appropriate warnings are generated and data processing continues where possible.

UDF integration testing executes Python functions in isolated subprocess environments with realistic data volumes to verify that memory usage, execution time, and error handling meet production requirements. We test both row-by-row UDF execution and batch processing modes to validate performance characteristics.

End-to-End Pipeline Execution

Complete pipeline integration tests execute full ETL workflows from source extraction through transformation to target loading. These tests use realistic data volumes (thousands to tens of thousands of records) to validate performance characteristics and identify bottlenecks that only appear under load.

We test various pipeline topologies including linear pipelines, diamond-shaped dependency graphs, and complex multi-source pipelines that combine data from multiple sources. Each test validates that data lineage tracking captures complete provenance information and that monitoring metrics accurately reflect pipeline performance.

Failure and Recovery Testing

Integration tests include comprehensive failure scenario testing to validate that our error handling and retry logic work correctly with real external systems. We use techniques like network manipulation, process termination, and resource exhaustion to simulate realistic failure conditions.

Recovery testing validates that pipelines can resume correctly after various failure types. We test scenarios including mid-pipeline failures, external system outages, and resource exhaustion, verifying that checkpointing and watermarking allow pipelines to resume from appropriate points without data duplication or loss.

The following table outlines key failure scenarios and their validation criteria:

Failure Type	Simulation Method	Recovery Validation
Database connection loss	Network partition during extraction	Resume from last successful batch
API rate limit exceeded	Mock service rate limiting	Automatic backoff and retry
Disk space exhaustion	Filesystem quota limits	Graceful failure with cleanup
Memory exhaustion	Large dataset processing	Streaming fallback or batch reduction
Process termination	SIGTERM during transformation	Checkpoint recovery on restart

Performance and Scalability Testing

Integration tests include performance benchmarks that validate system behavior under realistic load conditions. We test data processing throughput, memory usage patterns, and concurrent pipeline execution to identify performance bottlenecks and validate that resource usage scales appropriately with data volume.

Scalability testing validates that our system handles increasing data volumes gracefully. We test with datasets ranging from thousands to hundreds of thousands of records, measuring processing time, memory usage, and system resource consumption to identify scaling limitations.

Common Pitfalls in Integration Testing

⚠️ **Pitfall: Insufficient Test Data Realism** Using overly simplified test data fails to catch issues that occur with real-world data complexity. Create test datasets that include realistic data distributions, common data quality issues, and edge cases found in production systems. Include null values, special characters, and boundary values in test data.

⚠️ **Pitfall: Ignoring External System Variability** External systems behave differently under various conditions. Test with different response times, intermittent failures, and varying data volumes to ensure your pipeline handles real-world system variability. Don't assume external systems will always respond quickly or correctly.

⚠️ **Pitfall: Inadequate Cleanup Between Tests** ETL integration tests can leave persistent state in databases, file systems, and external services. Implement thorough cleanup procedures that run before and after each test to ensure test isolation and prevent cascading failures between test runs.

Milestone Checkpoints

Milestone 1: DAG Definition and Validation Checkpoint

After implementing the DAG definition and validation engine, the system must demonstrate correct parsing, validation, and visualization of pipeline definitions. This checkpoint validates that the foundation for all subsequent pipeline operations is solid and handles complex dependency scenarios correctly.

Validation Criteria for Milestone 1:

The checkpoint validates DAG parsing from both YAML and Python configuration formats, ensuring that all pipeline metadata including task configurations, dependencies, and parameters are correctly extracted. We test with increasingly complex pipeline definitions to verify that the system scales to realistic production scenarios.

Validation Area	Test Scenarios	Success Criteria
YAML Parsing	Simple linear pipeline, complex multi-source DAG	Correct task extraction and dependency mapping
Python Definition	Programmatic pipeline creation, dynamic task generation	Proper object model population
Cycle Detection	Intentional cycles, complex indirect cycles	Accurate cycle identification with path reporting
Topological Sort	Various DAG shapes, parallel execution opportunities	Correct execution levels with maximum parallelism
Parameter Substitution	Runtime variables, nested parameter references	Proper template resolution with type checking
Error Handling	Malformed YAML, invalid dependencies, missing tasks	Clear error messages with specific problem identification

Manual Verification Steps:

Execute the checkpoint validation by running a comprehensive test suite that exercises all DAG processing functionality. Start with simple pipeline definitions and progressively test more complex scenarios including pipelines with dozens of tasks and complex dependency relationships.

Verify DAG visualization by generating graph representations of test pipelines and confirming that the visual output accurately represents the dependency structure. Check that parallel execution opportunities are correctly identified and that critical path calculations provide accurate estimates.

Test error handling by intentionally creating invalid pipeline definitions and verifying that error messages provide sufficient detail for developers to identify and fix configuration problems.

Milestone 2: Data Extraction and Loading Checkpoint

The data extraction and loading checkpoint validates that connectors can reliably extract data from various sources and load it to target destinations with proper error handling and incremental processing capabilities. This milestone is critical because data connectivity

issues are among the most common ETL pipeline failures.

Source Connector Validation:

Database connector testing uses real database instances with controlled test schemas containing representative data types, null values, and edge cases. We validate that SQL query generation produces correct queries for various extraction patterns including simple SELECT statements, complex joins, and filtered extractions.

Incremental extraction testing verifies that watermarking correctly identifies changed records and that pagination handles large result sets efficiently. We test with datasets large enough to require multiple pages and verify that no records are missed or duplicated during extraction.

API connector testing validates HTTP client behavior including authentication, pagination, rate limiting, and error recovery. We test against both mock APIs and real external services where appropriate, ensuring that connector behavior matches API specifications and handles real-world API quirks.

The following validation criteria ensure connector reliability:

Connector Type	Validation Focus	Test Data Volume	Success Criteria
Database	SQL generation, type mapping, pagination	50K+ records	Complete extraction without data loss
REST API	Authentication, pagination, rate limiting	10K+ records across pages	Proper cursor handling and retry logic
File System	Format detection, streaming, compression	Multi-GB files	Memory-efficient processing

Destination Connector Validation:

Loading connector testing validates that data can be efficiently written to target systems with proper error handling and transactional guarantees. We test bulk loading scenarios with large datasets to verify that performance meets requirements and that partial failures are handled gracefully.

Schema mapping testing ensures that data types are correctly converted between source and destination systems, with appropriate warnings for lossy conversions and errors for incompatible types. We test with diverse data type combinations to verify comprehensive type system coverage.

Milestone 3: Data Transformation Checkpoint

The transformation checkpoint validates that both SQL-based transformations and Python UDFs execute correctly with realistic data volumes and handle error conditions gracefully. This milestone ensures that data quality and transformation logic meet business requirements.

SQL Transformation Validation:

SQL transformation testing executes template-based transformations against real database engines to verify that SQL generation produces correct results and handles parameter substitution properly. We test with complex transformations including aggregations, window functions, and multi-table joins.

Schema validation testing ensures that transformation outputs match expected schemas and that data quality rules correctly identify and handle invalid records. We test with datasets containing known data quality issues to verify that validation logic produces appropriate warnings and errors.

UDF Execution Validation:

Python UDF testing validates that user-defined functions execute correctly in isolated environments with proper resource management and error handling. We test both row-by-row processing and batch processing modes to verify performance characteristics meet requirements.

The following table outlines UDF validation criteria:

UDF Type	Execution Mode	Test Data	Success Criteria
Simple transformations	Row-by-row	100K records	< 10ms per record average
Complex analytics	Batch processing	1M+ records	Linear scaling with data volume
External API calls	Rate-limited batch	10K records	Proper throttling and error handling

Error Handling Validation:

Transformation error handling testing validates that invalid data is handled gracefully with appropriate logging and that pipeline execution continues where possible. We test scenarios including malformed input data, UDF exceptions, and resource exhaustion.

Milestone 4: Orchestration and Monitoring Checkpoint

The final checkpoint validates complete end-to-end pipeline execution with scheduling, monitoring, and alerting functionality. This milestone ensures that the system can reliably execute production workloads with appropriate observability and failure recovery.

Schedule Execution Validation:

Scheduler testing validates that pipelines execute according to configured schedules and that event-driven triggers work correctly. We test various cron expressions including edge cases like month boundaries and daylight saving time transitions.

Concurrent execution testing validates that the system handles multiple simultaneous pipeline runs correctly with proper resource isolation and conflict detection.

Monitoring and Alerting Validation:

Monitoring checkpoint testing validates that metrics are correctly collected and reported, that alerting triggers appropriately for various failure conditions, and that data lineage tracking captures complete provenance information.

We test monitoring under various load conditions to verify that metric collection doesn't significantly impact pipeline performance and that alerting provides actionable information for operators.

End-to-End Integration Validation:

The final validation executes complex multi-stage pipelines that exercise all system components together. These tests use realistic data volumes and complexity to validate that the complete system meets performance and reliability requirements.

We execute failure recovery testing that simulates various production failure scenarios and validates that the system recovers gracefully with minimal data loss or processing delays.

Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
Test Framework	pytest with fixtures	pytest + testcontainers + factory-boy
Mock Library	unittest.mock (built-in)	responses + freezegun + factory-boy
Database Testing	SQLite in-memory	PostgreSQL testcontainers
API Testing	requests-mock	WireMock or Prism mock server
Performance Testing	pytest-benchmark	locust for load testing
Test Data	JSON fixtures	factory-boy data factories

Recommended Test Structure:

```
project-root/
tests/
  unit/                          ← isolated component tests
    test_dag_engine.py           ← DAG parsing and validation
    test_connectors.py           ← individual connector logic
    test_transformations.py      ← transformation functions
    test_scheduler.py            ← scheduling logic
    test_state_machine.py        ← state transition logic
  integration/                   ← end-to-end component tests
    test_pipeline_execution.py   ← complete pipeline runs
    test_external_systems.py     ← real database/API tests
    test_failure_recovery.py     ← error handling scenarios
  milestones/                    ← milestone checkpoint tests
    test_milestone_1.py          ← DAG definition checkpoint
    test_milestone_2.py          ← extraction/loading checkpoint
    test_milestone_3.py          ← transformation checkpoint
    test_milestone_4.py          ← orchestration checkpoint
  fixtures/                      ← test data and configuration
    pipelines/                   ← sample pipeline definitions
    data/                        ← test datasets
    mocks/                       ← mock service configurations
  conftest.py                    ← pytest configuration and shared fixtures
```

Unit Test Infrastructure Code:

PYTHON

```python
# tests/conftest.py

import pytest

import tempfile

import shutil

from unittest.mock import MagicMock

from datetime import datetime, timedelta

from typing import Dict, Any, List


@pytest.fixture

def mock_database_connection():

    """Mock database connection for unit tests."""

    conn = MagicMock()

    conn.execute.return_value = []

    conn.fetchall.return_value = []

    conn.fetchone.return_value = None

    conn.commit.return_value = None

    conn.rollback.return_value = None

    conn.close.return_value = None

    return conn


@pytest.fixture

def mock_http_client():

    """Mock HTTP client for API connector tests."""

    client = MagicMock()

    client.get.return_value.status_code = 200

    client.get.return_value.json.return_value = {"data": []}

    client.post.return_value.status_code = 201

    return client


@pytest.fixture

def sample_pipeline_definition():

    """Standard pipeline definition for testing."""

    return PipelineDefinition(

        id="test-pipeline",
```

```
        name="Test Pipeline",
        description="Pipeline for unit testing",
        schedule="0 * * * *",
        tasks=[
            TaskDefinition(
                id="extract-task",
                name="Extract Data",
                type="database_extract",
                config={"query": "SELECT * FROM users"},
                dependencies=[],
                retry_policy=RetryPolicy(max_attempts=3, backoff_seconds=60, exponential_backoff=True,
retry_on_error_types=["NETWORK_ERROR"]),
                timeout_seconds=300
            ),
            TaskDefinition(
                id="transform-task",
                name="Transform Data",
                type="sql_transform",
                config={"query": "SELECT id, UPPER(name) as name FROM input"},
                dependencies=["extract-task"],
                retry_policy=RetryPolicy(max_attempts=2, backoff_seconds=30, exponential_backoff=False,
retry_on_error_types=["DATA_QUALITY_ERROR"]),
                timeout_seconds=600
            )
        ],
        parameters={"source_table": "users"},
        created_at=datetime.now(),
        version=1
    )

@pytest.fixture
def temp_directory():
    """Temporary directory for file-based tests."""
    temp_dir = tempfile.mkdtemp()
```

```
    yield temp_dir

    shutil.rmtree(temp_dir)


class MockTimeProvider:

    """Controllable time source for testing scheduling logic."""



    def __init__(self, start_time: datetime):

        self.current_time = start_time



    def now(self) -> datetime:

        return self.current_time



    def advance(self, delta: timedelta):

        self.current_time += delta


@pytest.fixture

def mock_time():

    """Mock time provider for deterministic scheduling tests."""

    return MockTimeProvider(datetime(2024, 1, 1, 12, 0, 0))
```

Unit Test Skeleton Code:

PYTHON

```python
# tests/unit/test_dag_engine.py

import pytest

from dag_engine import DAGEngine, CycleDetectionResult, ExecutionPlan

from data_model import PipelineDefinition, TaskDefinition


class TestDAGValidation:

    """Test suite for DAG parsing and validation logic."""



    def test_detect_cycles_simple_cycle(self):

        """Test cycle detection with a simple two-node cycle."""

        # TODO 1: Create adjacency list representing A -> B -> A cycle

        # TODO 2: Call detect_cycles_dfs with the adjacency list

        # TODO 3: Assert that has_cycle is True

        # TODO 4: Assert that cycle_path contains the expected cycle nodes

        # TODO 5: Verify that cycle_path forms a valid cycle (first == last node)

        pass



    def test_topological_sort_parallel_tasks(self):

        """Test topological sort identifies parallel execution opportunities."""

        # TODO 1: Create adjacency list with diamond dependency pattern (A -> B,C -> D)

        # TODO 2: Call topological_sort_kahns with the adjacency list

        # TODO 3: Assert that result has correct number of execution levels

        # TODO 4: Assert that B and C are in the same execution level (can run parallel)

        # TODO 5: Assert that A is in first level and D is in last level

        pass



    def test_parse_yaml_invalid_syntax(self):

        """Test YAML parsing with malformed syntax."""

        # TODO 1: Create YAML string with invalid syntax (missing colons, wrong indentation)

        # TODO 2: Call parse_yaml_file with the invalid YAML

        # TODO 3: Assert that appropriate exception is raised

        # TODO 4: Verify that error message contains useful debugging information

        pass
```

```python
class TestExecutionPlanning:
    """Test suite for DAG execution planning logic."""


    def test_calculate_critical_path(self):
        """Test critical path calculation with task duration estimates."""
        # TODO 1: Create adjacency list with known task dependencies
        # TODO 2: Create task_durations dict with estimated execution times
        # TODO 3: Call calculate_critical_path with adjacency list and durations
        # TODO 4: Assert that returned path represents the longest duration chain
        # TODO 5: Assert that total duration matches sum of critical path task durations
        pass
```

Integration Test Infrastructure:

PYTHON

```python
# tests/integration/conftest.py

import pytest

import docker

import time

import psycopg2

from testcontainers.postgres import PostgresContainer

from testcontainers.compose import DockerCompose


@pytest.fixture(scope="session")

def postgres_container():

    """PostgreSQL container for integration testing."""

    with PostgresContainer("postgres:13", driver="psycopg2") as postgres:

        # Wait for container to be ready

        time.sleep(5)



        # Create test schema and sample data

        connection = psycopg2.connect(postgres.get_connection_url())

        with connection.cursor() as cursor:

            cursor.execute("""

                CREATE TABLE users (

                    id SERIAL PRIMARY KEY,

                    name VARCHAR(100) NOT NULL,

                    email VARCHAR(100) UNIQUE,

                    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

                    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP

                );



                INSERT INTO users (name, email) VALUES

                ('Alice Johnson', 'alice@example.com'),

                ('Bob Smith', 'bob@example.com'),

                ('Charlie Brown', 'charlie@example.com');

            """)

            connection.commit()
```

```
        yield postgres

@pytest.fixture(scope="session")
def test_environment():
    """Complete test environment with all external services."""
    with DockerCompose("tests/integration", compose_file_name="docker-compose.test.yml") as compose:
        # Wait for all services to be ready
        time.sleep(10)
        yield compose

class IntegrationTestHelper:
    """Helper utilities for integration testing."""

    @staticmethod
    def wait_for_pipeline_completion(pipeline_run_id: str, timeout_seconds: int = 300):
        """Wait for pipeline run to complete with timeout."""
        # TODO: Implement polling logic to check pipeline run status
        # TODO: Raise TimeoutError if pipeline doesn't complete within timeout
        pass

    @staticmethod
    def validate_data_lineage(source_table: str, target_table: str, expected_transformations: List[str]):
        """Validate that data lineage correctly captures transformations."""
        # TODO: Query lineage tracking system
        # TODO: Verify that all expected transformations are recorded
        # TODO: Assert that source and target tables are correctly linked
        pass
```

Milestone Checkpoint Implementation:

PYTHON

```python
# tests/milestones/test_milestone_1.py

import pytest

import yaml

from pathlib import Path

from dag_engine import DAGEngine

from data_model import PipelineDefinition


class TestMilestone1Checkpoint:

    """Milestone 1: DAG Definition and Validation checkpoint tests."""


    def test_yaml_pipeline_parsing(self):

        """Verify YAML pipeline definitions parse correctly."""

        # TODO 1: Load sample YAML pipeline from fixtures

        # TODO 2: Parse YAML using parse_yaml_file function

        # TODO 3: Validate all required fields are present in PipelineDefinition

        # TODO 4: Verify task dependencies are correctly mapped

        # TODO 5: Check that parameters are properly extracted

        pass


    def test_complex_dag_validation(self):

        """Test validation with realistic complex pipeline."""

        # TODO 1: Create pipeline with 10+ tasks and complex dependencies

        # TODO 2: Validate that cycle detection completes successfully

        # TODO 3: Verify topological sort produces valid execution order

        # TODO 4: Check that parallel execution opportunities are identified

        # TODO 5: Validate that critical path calculation is accurate

        pass


    def test_error_reporting_quality(self):

        """Verify that validation errors provide actionable information."""

        # TODO 1: Create pipeline with intentional errors (cycles, missing tasks)

        # TODO 2: Run validation and capture error messages

        # TODO 3: Assert that errors specify exact problem location
```

```
        # TODO 4: Verify that suggested fixes are included where possible
        pass

# Checkpoint validation command:
# pytest tests/milestones/test_milestone_1.py -v
# Expected: All tests pass, demonstrating working DAG engine
```

Debugging Tips:

Symptom	Likely Cause	Diagnosis Steps	Fix
Tests hang indefinitely	External service dependency not mocked	Check test for database/API calls	Add appropriate mocks or use testcontainers
Intermittent test failures	Race conditions in concurrent code	Run tests multiple times, check for shared state	Add proper synchronization or test isolation
Memory usage grows during tests	Test data not cleaned between runs	Monitor memory usage with memory_profiler	Implement proper teardown in fixtures
Integration tests fail in CI	Different environment configuration	Compare local vs CI environment settings	Standardize environment using containers

Milestone Checkpoint Commands:

After completing each milestone, run these validation commands:

```bash
# Milestone 1: DAG Definition and Validation
pytest tests/milestones/test_milestone_1.py -v
python -m dag_engine.visualizer --input tests/fixtures/pipelines/complex.yaml --output /tmp/dag.svg

# Milestone 2: Data Extraction and Loading
pytest tests/milestones/test_milestone_2.py -v
pytest tests/integration/test_connectors.py -v --tb=short

# Milestone 3: Data Transformations
pytest tests/milestones/test_milestone_3.py -v
pytest tests/integration/test_transformations.py -v

# Milestone 4: Orchestration and Monitoring
pytest tests/milestones/test_milestone_4.py -v
pytest tests/integration/test_pipeline_execution.py -v

# Complete test suite
pytest tests/ -v --cov=src --cov-report=html
```

Expected behavior after each milestone:

- All milestone-specific tests pass without errors
- Integration tests demonstrate working functionality with real data
- Performance benchmarks meet established criteria
- Manual verification steps complete successfully
- System demonstrates graceful error handling and recovery

Debugging Guide

> **Milestone(s):** All milestones - systematic debugging approaches are essential for pipeline definition troubleshooting (Milestone 1), data processing issue resolution (Milestones 2-3), and orchestration debugging (Milestone 4).

Mental Model: Medical Diagnosis

Think of debugging ETL pipelines like being a doctor diagnosing a patient. You start by gathering symptoms (error messages, performance metrics, unusual behaviors), then form hypotheses about potential causes based on your understanding of how the system works. You run specific tests to confirm or rule out each hypothesis, narrowing down to the root cause. Just as a doctor has a systematic approach to diagnosis - checking vital signs first, then running targeted tests - debugging requires a methodical approach rather than random guessing.

The key insight is that symptoms often manifest far from their root causes in distributed systems. A task timeout might be caused by network congestion, resource exhaustion, or a deadlock in a completely different component. Like referred pain in medicine, the symptom location doesn't always indicate the problem source.

Common Symptoms and Causes

ETL pipeline debugging follows predictable patterns. Most issues fall into a few categories with characteristic symptoms that can guide diagnosis. Understanding these symptom-cause patterns allows developers to quickly narrow the search space and apply targeted fixes.

The following symptom-cause mapping represents the most common issues encountered in production ETL systems, organized by the component where symptoms first appear:

Symptom	Likely Cause	Immediate Diagnosis Steps	Fix Strategy
Pipeline stuck in PENDING state	DAG parsing errors or validation failures	Check pipeline definition YAML/Python syntax; examine validation error logs	Fix syntax errors, resolve dependency cycles, validate task configurations
Tasks skip unexpectedly	Upstream dependencies failed or wrong dependency configuration	Trace upstream task states; verify dependency specifications match actual task IDs	Correct dependency declarations, fix upstream failures, check for typos in task IDs
"Cycle detected" error during parsing	Circular dependencies in task definitions	Run `detect_cycles_dfs()` manually on pipeline definition; examine dependency graph visualization	Remove circular dependencies, consider splitting tasks, add conditional logic
Tasks run in wrong order	Topological sort error or missing dependencies	Compare actual execution order with expected DAG visualization; check for missing dependency declarations	Add missing dependencies, verify topological sort implementation, check for race conditions
Database connection timeouts	Connection pool exhaustion or network issues	Monitor active connections, check network latency, examine connection pool metrics	Increase pool size, implement connection retry logic, optimize query performance
API extraction fails intermittently	Rate limiting, authentication expiry, or pagination issues	Check API response headers for rate limit info; examine authentication token validity; verify pagination cursor handling	Implement exponential backoff, refresh tokens proactively, fix pagination logic
Incremental loads miss data	Watermark not updated atomically or clock skew issues	Verify watermark update happens in same transaction as data load; check for time zone mismatches	Use database transactions for atomic updates, add lookback window for clock skew
Duplicate records in destination	Upsert logic failures or non-atomic operations	Check for unique constraint violations; examine upsert key definitions; verify transaction isolation	Fix upsert key selection, use proper transaction boundaries, implement deduplication
Schema validation errors	Type coercion failures or schema evolution issues	Compare source and destination schemas; check for null handling differences; examine type conversion logs	Update schema mappings, implement graceful type coercion, handle null value differences
Transformation timeouts	Memory exhaustion from large datasets or inefficient SQL	Monitor memory usage during transformation; examine SQL execution plans; check for full table scans	Implement streaming processing, optimize SQL queries, add memory limits and checkpoints
Python UDF crashes	Unhandled exceptions or resource limits exceeded	Examine UDF execution logs; check memory and CPU usage; verify error handling in UDF code	Add try-catch blocks, implement resource monitoring, use subprocess isolation
Data quality validation failures	Source data corruption or business rule violations	Sample failed records; examine validation rule logic; check for data drift in source systems	Implement data profiling, adjust validation thresholds, add data quality monitoring
Pipeline runs never start	Scheduler configuration errors or resource unavailability	Check cron expression syntax; verify scheduler service health; examine resource allocation logs	Fix schedule syntax, restart scheduler service, increase resource limits

Symptom	Likely Cause	Immediate Diagnosis Steps	Fix Strategy
Tasks hang in RUNNING state indefinitely	Deadlocks, infinite loops, or resource starvation	Check for database locks; examine task execution logs; monitor CPU and memory usage	Implement task timeouts, add deadlock detection, optimize resource usage
Pipeline fails but tasks show SUCCESS	State management race conditions or incomplete failure propagation	Compare task-level and pipeline-level state; check for concurrent state updates; examine event ordering	Fix state transition logic, implement proper locking, ensure atomic state updates
Memory usage grows continuously	Memory leaks in transformation code or unclosed connections	Profile memory usage over time; check for unclosed database connections; examine object lifecycle	Fix connection management, implement proper cleanup, add garbage collection monitoring
High CPU usage with no progress	Infinite loops, busy waiting, or inefficient algorithms	Profile CPU usage by component; examine algorithm complexity; check for blocking operations	Optimize algorithms, add sleep to polling loops, implement proper backpressure
Disk space fills rapidly	Excessive logging, temporary file accumulation, or large intermediate results	Check log file sizes; examine temporary directory usage; monitor intermediate data volumes	Implement log rotation, clean up temporary files, use streaming for large datasets
Network timeouts between components	Network congestion, firewall issues, or component overload	Check network latency and packet loss; verify firewall rules; examine component health metrics	Increase timeout values, implement retry logic, optimize network usage

> **Key Insight**: The vast majority of ETL pipeline issues stem from three root causes: resource management problems (memory, connections, disk), state consistency issues (race conditions, atomic operations), and data quality problems (schema changes, null handling). Understanding these patterns helps focus debugging efforts.

Common Anti-Patterns and Debugging Traps

⚠️ **Pitfall: Log Flooding During Debugging** Developers often enable verbose logging across all components when debugging, generating massive log volumes that obscure the actual problem. This makes debugging slower and can impact system performance.

Why it's wrong: Excessive logging creates noise that hides signal, fills disk space rapidly, and can cause performance degradation that masks the original issue.

Fix: Enable targeted logging only for the specific component and time window where the issue occurs. Use log levels strategically and implement log sampling for high-frequency events.

⚠️ **Pitfall: Testing in Isolation Only** Testing individual components in isolation without integration testing often misses issues that only appear when components interact under realistic conditions.

Why it's wrong: Many ETL issues emerge from component interactions, timing dependencies, and resource contention that don't appear in unit tests.

Fix: Implement comprehensive integration tests with realistic data volumes and concurrent execution patterns. Test failure scenarios and recovery paths.

⚠️ **Pitfall: Ignoring Resource Limits** Debugging in development environments without realistic resource constraints often fails to reveal issues that only appear under production load.

Why it's wrong: Memory leaks, connection pool exhaustion, and performance degradation only manifest under realistic load conditions.

Fix: Use production-like resource limits in testing environments. Implement resource monitoring and alerting to catch issues early.

Debugging Techniques

Effective ETL debugging requires a systematic approach that combines multiple investigation techniques. The key is to gather evidence methodically rather than making assumptions about where problems might be.

Log-Based Investigation

Pipeline debugging starts with understanding the log structure and using logs strategically to trace execution flow and identify anomalies.

Log Correlation Strategy

ETL systems generate logs from multiple components, making it challenging to trace a single pipeline run across the system. Implement correlation using the pipeline run ID and task execution ID to connect related log entries:

1. Start with the pipeline-level logs to understand the overall execution state and timeline
2. Identify which tasks failed or behaved unexpectedly based on pipeline logs
3. Drill down to task-level logs using the task execution ID to examine detailed behavior
4. Follow data lineage through transformation logs to understand data flow issues
5. Correlate with system-level logs (database, message broker) using timestamps and correlation IDs

Log Analysis Patterns

Log Pattern	Indicates	Investigation Steps
Gaps in timestamp sequence	Component hang or crash	Check system logs for crash dumps; examine memory and CPU usage during gap period
Repeated error messages with same context	Retry loop without progress	Examine retry policy configuration; check if underlying issue is being addressed
Error messages without stack traces	Swallowed exceptions	Review error handling code; ensure proper exception logging and propagation
Performance metrics showing degradation over time	Resource leak or memory pressure	Profile memory usage; check for unclosed connections or accumulating objects
Inconsistent state between components	Race condition or incomplete transaction	Check transaction boundaries; examine concurrent access patterns

Structured Log Querying

Implement structured logging with consistent field names to enable efficient querying:

```
timestamp=2024-01-15T10:30:45Z level=ERROR component=task-executor
pipeline_id=pipeline-123 run_id=run-456 task_id=extract-customers
attempt=2 error_type=TIMEOUT_ERROR message="Database query timeout after 300s"
```

Use log aggregation tools to query across multiple components and time ranges. Key queries for ETL debugging:

- Find all errors for a specific pipeline run: `run_id=run-456 AND level=ERROR`
- Trace task execution timeline: `task_id=extract-customers ORDER BY timestamp`
- Identify resource exhaustion patterns: `error_type=RESOURCE_EXHAUSTION last 24h`
- Monitor retry patterns: `attempt>1 GROUP BY error_type`

State Inspection and Monitoring

Understanding current system state is crucial for diagnosing issues, especially for intermittent problems that don't leave clear log traces.

Database State Analysis

The pipeline metadata database contains the authoritative state for all executions. Use direct database queries to understand state inconsistencies:

State Query Purpose	SQL Pattern	Key Information
Find stuck pipelines	`SELECT * FROM pipeline_runs WHERE state='RUNNING' AND started_at < NOW() - INTERVAL '2 hours'`	Long-running executions that may be stuck
Identify retry patterns	`SELECT task_id, COUNT(*) as attempts FROM task_executions WHERE pipeline_run_id='run-456' GROUP BY task_id`	Tasks requiring multiple attempts
Check dependency resolution	`SELECT upstream_task, downstream_task FROM task_dependencies WHERE pipeline_id='pipeline-123'`	Verify dependencies match expectations
Monitor failure rates	`SELECT DATE(completed_at), state, COUNT(*) FROM pipeline_runs GROUP BY DATE(completed_at), state`	Track success/failure trends over time

Resource Monitoring Integration

Correlate application metrics with system-level resource usage to identify bottlenecks:

- **Memory patterns**: Track heap usage during transformation tasks to identify memory leaks
- **Connection pools**: Monitor active/idle connections to detect pool exhaustion
- **Disk I/O**: Watch for disk space consumption during large data loads
- **Network utilization**: Monitor bandwidth usage during data extraction phases
- **CPU usage**: Identify compute-intensive transformations that need optimization

Interactive Debugging Techniques

For complex issues that require real-time investigation, interactive debugging provides deeper insights than static log analysis.

Pipeline Replay and Simulation

Implement pipeline replay functionality to reproduce issues in controlled environments:

1. Capture pipeline state and input data from the failed execution
2. Create isolated replay environment with same resource constraints
3. Execute pipeline with additional debugging instrumentation enabled
4. Step through execution phases to identify exact failure point
5. Modify inputs systematically to isolate root cause

Live State Inspection

Provide debugging APIs that allow real-time inspection of running pipelines:

Debug API Endpoint	Purpose	Usage
`/debug/pipeline/{run_id}/state`	Current state of all tasks	Check for unexpected task states or timing issues
`/debug/task/{execution_id}/metrics`	Real-time task metrics	Monitor memory, CPU, and data throughput during execution
`/debug/connections/pools`	Connection pool status	Identify connection leaks or pool exhaustion
`/debug/queues/depths`	Message queue depths	Detect backpressure or component overload

Component Health Checks

Implement comprehensive health checks that validate both basic connectivity and operational readiness:

- **Database connectivity**: Verify read/write access with sample queries
- **External API availability**: Test authentication and basic endpoint access
- **Message broker health**: Confirm topic access and message flow
- **Resource availability**: Check disk space, memory, and CPU capacity
- **Schema compatibility**: Validate source/destination schema compatibility

Performance Debugging

Performance issues in ETL pipelines often manifest as gradually degrading throughput, increasing memory usage, or extended execution times. These issues require specialized debugging approaches that focus on resource utilization and algorithmic efficiency.

Identifying Performance Bottlenecks

Performance debugging starts with establishing baseline metrics and identifying deviations from expected behavior patterns.

Throughput Analysis

ETL pipeline performance is primarily measured in terms of data throughput and execution time. Establish baseline metrics for comparison:

Performance Metric	Measurement Method	Typical Issues	Investigation Steps
Records per second processed	Count input/output records divided by execution time	Decreasing over time, varies between runs	Check for data volume changes, schema complexity increases, resource contention
Data volume throughput	Bytes processed per minute during extraction/loading	Network bandwidth limitations, serialization overhead	Monitor network utilization, examine data compression, check serialization efficiency
Task execution time	Wall clock time from start to completion	Individual tasks taking longer than expected	Profile task internals, check for blocking operations, examine algorithm complexity
End-to-end pipeline latency	Time from trigger to completion	Overall pipeline slowdown affecting SLAs	Analyze critical path, identify parallelization opportunities, check for sequential bottlenecks

Resource Utilization Patterns

Different types of performance issues create characteristic resource usage patterns:

- **Memory-bound operations**: Steady memory growth during execution, potential out-of-memory errors
- **CPU-bound operations**: High CPU utilization with low I/O, often in transformation phases
- **I/O-bound operations**: High disk or network activity with lower CPU usage
- **Lock contention**: Low resource usage but poor throughput due to blocking

Memory Performance Issues

Memory-related performance problems are common in ETL pipelines due to large dataset processing and transformation operations.

Memory Leak Detection

Memory leaks in ETL pipelines typically occur in transformation code, connection management, or intermediate result caching:

1. **Monitor heap growth patterns** over multiple pipeline executions to identify consistent memory increases
2. **Profile object allocation** during transformation phases to identify accumulating objects
3. **Check connection lifecycle** to ensure database connections are properly closed
4. **Examine caching behavior** for intermediate results that may not be evicted properly

Memory Optimization Strategies

Memory Issue	Symptoms	Solution Approach
Large dataset transformations	Out-of-memory during SQL operations	Implement streaming processing, use temp tables for intermediate results
Python UDF memory accumulation	Memory usage grows within single task	Add explicit garbage collection, use generator functions for large datasets
Connection object accumulation	Memory growth proportional to connection usage	Implement proper connection pooling, ensure connections are returned to pool
Intermediate result caching	Memory usage doesn't decrease between pipeline runs	Implement cache eviction policies, use disk-based caching for large results

Database Performance Debugging

Database operations often become bottlenecks in ETL pipelines, especially during bulk loading and complex transformations.

Query Performance Analysis

Database performance issues require examining both query structure and execution patterns:

1. **Capture query execution plans** for all SQL operations to identify inefficient operations
2. **Monitor query execution times** and correlate with data volume changes
3. **Check for missing indexes** on frequently queried columns, especially join and filter columns
4. **Analyze transaction isolation levels** to prevent lock contention issues

Bulk Operation Optimization

ETL pipelines perform many bulk operations that require specific optimization approaches:

Operation Type	Performance Consideration	Optimization Strategy
Bulk inserts	Insert speed vs. transaction safety	Use batch inserts with optimal batch size, disable non-essential indexes during load
Upsert operations	Conflict detection overhead	Implement efficient merge strategies, use staging tables for large upserts
Large result sets	Memory usage and network transfer	Stream results using cursor-based pagination, implement result set compression
Complex transformations	CPU usage and temp space	Break complex operations into smaller steps, use appropriate join algorithms

Network and I/O Performance

ETL pipelines often move large amounts of data across network connections, making network and I/O performance critical.

Network Bottleneck Identification

Network performance issues manifest in several ways:

- **High latency**: Individual operations take longer but overall bandwidth is acceptable
- **Low throughput**: Network bandwidth is saturated, affecting overall pipeline performance
- **Intermittent failures**: Network instability causing connection drops and retries

I/O Optimization Techniques

File and database I/O optimization focuses on minimizing the number of operations and maximizing transfer efficiency:

1. **Use connection pooling** to avoid connection setup overhead for database operations
2. **Implement read-ahead buffering** for sequential file operations
3. **Use compression** for network transfers when CPU resources allow
4. **Batch operations** to reduce the overhead of individual I/O calls
5. **Implement parallel I/O** for independent operations that can run concurrently

> **Performance Debugging Principle**: Always measure before optimizing. Many performance "improvements" actually make things worse by optimizing the wrong bottleneck or introducing complexity that reduces maintainability without providing measurable benefits.

Implementation Guidance

The debugging infrastructure should be built into the system from the beginning rather than added as an afterthought. This section provides concrete tools and techniques for implementing effective debugging capabilities.

Technology Recommendations for Debugging

Component	Simple Option	Advanced Option
Log Aggregation	File-based logging with log rotation	ELK Stack (Elasticsearch, Logstash, Kibana) or cloud logging services
Metrics Collection	Prometheus with Grafana	DataDog, New Relic, or comprehensive APM solutions
Distributed Tracing	Correlation IDs in logs	OpenTelemetry with Jaeger or Zipkin
Performance Profiling	Built-in Python profiler	Continuous profiling with Pyflame or py-spy
Database Monitoring	Query logging and pg_stat_statements	Dedicated database monitoring tools

Recommended File Structure for Debugging Tools

```
project-root/
  debugging/
    profiling/
      memory_profiler.py      ← Memory usage analysis
      cpu_profiler.py         ← CPU performance profiling
    monitoring/
      health_checks.py        ← Component health verification
      metrics_collector.py    ← Performance metrics gathering
    log_analysis/
      log_parser.py          ← Structured log parsing
      correlation.py         ← Cross-component log correlation
  tests/
    debugging/
      test_profiling.py      ← Profiling tool tests
      test_monitoring.py     ← Monitoring functionality tests
  scripts/
    debug_pipeline.py        ← Interactive pipeline debugging script
    replay_pipeline.py       ← Pipeline execution replay utility
```

Essential Debugging Infrastructure

PYTHON

```python
# debugging/monitoring/health_checks.py

from typing import Dict, List, Optional

from dataclasses import dataclass

from enum import Enum

import time

import psutil

import psycopg2


class HealthStatus(Enum):

    HEALTHY = "healthy"

    DEGRADED = "degraded"

    UNHEALTHY = "unhealthy"


@dataclass

class HealthCheckResult:

    component: str

    status: HealthStatus

    message: str

    metrics: Dict[str, float]

    timestamp: float


class SystemHealthChecker:

    """Comprehensive health checking for ETL pipeline components."""



    def __init__(self, db_connection_string: str, message_broker_url: str):

        self.db_connection_string = db_connection_string

        self.message_broker_url = message_broker_url



    def check_system_resources(self) -> HealthCheckResult:

        """Check basic system resource availability."""

        try:

            memory = psutil.virtual_memory()

            disk = psutil.disk_usage('/')
```

```
            cpu_percent = psutil.cpu_percent(interval=1)


            metrics = {
                'memory_usage_percent': memory.percent,
                'disk_usage_percent': (disk.used / disk.total) * 100,
                'cpu_usage_percent': cpu_percent,
                'available_memory_mb': memory.available / (1024 * 1024)
            }


            # Determine status based on thresholds
            if memory.percent > 90 or disk.used / disk.total > 0.95:
                status = HealthStatus.UNHEALTHY
                message = f"Critical resource usage: Memory {memory.percent}%, Disk
{(disk.used/disk.total)*100:.1f}%"
            elif memory.percent > 80 or disk.used / disk.total > 0.85:
                status = HealthStatus.DEGRADED
                message = f"High resource usage: Memory {memory.percent}%, Disk
{(disk.used/disk.total)*100:.1f}%"
            else:
                status = HealthStatus.HEALTHY
                message = "System resources normal"


            return HealthCheckResult(
                component="system_resources",
                status=status,
                message=message,
                metrics=metrics,
                timestamp=time.time()
            )


        except Exception as e:
            return HealthCheckResult(
                component="system_resources",
                status=HealthStatus.UNHEALTHY,
```

```
            message=f"Failed to check system resources: {str(e)}",
            metrics={},
            timestamp=time.time()
        )

def check_database_connectivity(self) -> HealthCheckResult:
    """Verify database connection and basic operations."""
    start_time = time.time()
    try:
        conn = psycopg2.connect(self.db_connection_string)
        cursor = conn.cursor()

        # Test basic read operation
        cursor.execute("SELECT 1")
        result = cursor.fetchone()

        # Test write operation to a health check table
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS health_check_temp (
                check_time TIMESTAMP DEFAULT NOW()
            )
        """)
        cursor.execute("INSERT INTO health_check_temp DEFAULT VALUES")

        # Clean up
        cursor.execute("DELETE FROM health_check_temp WHERE check_time < NOW() - INTERVAL '1 hour'")
        conn.commit()

        response_time = (time.time() - start_time) * 1000  # Convert to milliseconds

        cursor.close()
        conn.close()
```

```
            metrics = {'response_time_ms': response_time}

            if response_time > 5000:  # 5 seconds
                status = HealthStatus.DEGRADED
                message = f"Database responding slowly: {response_time:.1f}ms"
            else:
                status = HealthStatus.HEALTHY
                message = f"Database healthy: {response_time:.1f}ms response"

            return HealthCheckResult(
                component="database",
                status=status,
                message=message,
                metrics=metrics,
                timestamp=time.time()
            )

        except Exception as e:
            return HealthCheckResult(
                component="database",
                status=HealthStatus.UNHEALTHY,
                message=f"Database connection failed: {str(e)}",
                metrics={'response_time_ms': (time.time() - start_time) * 1000},
                timestamp=time.time()
            )

# debugging/log_analysis/correlation.py
import re
import json
from typing import Dict, List, Optional
from dataclasses import dataclass
from datetime import datetime

@dataclass
```

```
class LogEntry:

    timestamp: datetime

    level: str

    component: str

    pipeline_id: Optional[str]

    run_id: Optional[str]

    task_id: Optional[str]

    message: str

    raw_line: str


class LogCorrelator:

    """Correlate logs across components for pipeline debugging."""



    def __init__(self):

        # Regex pattern for structured log format

        self.log_pattern = re.compile(

            r'timestamp=(?P<timestamp>\S+)\s+'

            r'level=(?P<level>\S+)\s+'

            r'component=(?P<component>\S+)'

            r'(?:\s+pipeline_id=(?P<pipeline_id>\S+))?'

            r'(?:\s+run_id=(?P<run_id>\S+))?'

            r'(?:\s+task_id=(?P<task_id>\S+))?'

            r'.*message="(?P<message>[^"]*)"'

        )



    def parse_log_entry(self, log_line: str) -> Optional[LogEntry]:

        """Parse structured log entry from log line."""

        match = self.log_pattern.match(log_line.strip())

        if not match:

            return None



        try:

            timestamp = datetime.fromisoformat(match.group('timestamp').replace('Z', '+00:00'))
```

```
        except ValueError:
            return None

        return LogEntry(
            timestamp=timestamp,
            level=match.group('level'),
            component=match.group('component'),
            pipeline_id=match.group('pipeline_id'),
            run_id=match.group('run_id'),
            task_id=match.group('task_id'),
            message=match.group('message'),
            raw_line=log_line
        )

    def correlate_pipeline_logs(self, log_entries: List[LogEntry], run_id: str) -> Dict[str, List[LogEntry]]:
        """Group log entries by component for a specific pipeline run."""
        correlated_logs = {}

        for entry in log_entries:
            if entry.run_id == run_id:
                component = entry.component
                if component not in correlated_logs:
                    correlated_logs[component] = []
                correlated_logs[component].append(entry)

        # Sort each component's logs by timestamp
        for component in correlated_logs:
            correlated_logs[component].sort(key=lambda x: x.timestamp)

        return correlated_logs

    def find_error_context(self, log_entries: List[LogEntry], error_entry: LogEntry,
                           context_seconds: int = 30) -> List[LogEntry]:
```

```
        """Find log entries around an error for context."""
        error_time = error_entry.timestamp
        context_logs = []

        for entry in log_entries:
            # Same run and within time window
            if (entry.run_id == error_entry.run_id and
                abs((entry.timestamp - error_time).total_seconds()) <= context_seconds):
                context_logs.append(entry)

        return sorted(context_logs, key=lambda x: x.timestamp)
```

Core Debugging Tools Implementation

```python
# debugging/profiling/performance_analyzer.py

import time

import threading

import psutil

from collections import defaultdict, deque

from typing import Dict, List, Optional


class PerformanceAnalyzer:

    """Real-time performance monitoring and analysis."""


    def __init__(self, sample_interval_seconds: float = 1.0):

        self.sample_interval = sample_interval_seconds

        self.monitoring = False

        self.samples = defaultdict(deque)

        self.monitor_thread = None


    def start_monitoring(self, pipeline_run_id: str):

        """Begin performance monitoring for a pipeline run."""

        self.monitoring = True

        self.current_run_id = pipeline_run_id

        self.monitor_thread = threading.Thread(

            target=self._monitoring_loop,

            name=f"perf-monitor-{pipeline_run_id}"

        )

        self.monitor_thread.daemon = True

        self.monitor_thread.start()


    def stop_monitoring(self) -> Dict[str, List[float]]:

        """Stop monitoring and return collected samples."""

        self.monitoring = False

        if self.monitor_thread:

            self.monitor_thread.join(timeout=5.0)
```

```
        # Convert deques to lists for JSON serialization
        results = {}
        for metric_name, samples in self.samples.items():
            results[metric_name] = list(samples)

        # Clear samples for next run
        self.samples.clear()
        return results

    def _monitoring_loop(self):
        """Background thread that collects performance samples."""
        while self.monitoring:
            timestamp = time.time()

            # Collect system metrics
            memory = psutil.virtual_memory()
            cpu_percent = psutil.cpu_percent()

            # Store samples with timestamp
            self.samples['timestamp'].append(timestamp)
            self.samples['memory_usage_mb'].append(memory.used / (1024 * 1024))
            self.samples['memory_percent'].append(memory.percent)
            self.samples['cpu_percent'].append(cpu_percent)

            # Collect process-specific metrics
            try:
                process = psutil.Process()
                process_memory = process.memory_info()
                self.samples['process_memory_mb'].append(process_memory.rss / (1024 * 1024))
                self.samples['process_cpu_percent'].append(process.cpu_percent())
            except psutil.Error:
                pass
```

```
            time.sleep(self.sample_interval)

# scripts/debug_pipeline.py
#!/usr/bin/env python3
"""Interactive pipeline debugging utility."""

import sys
import json
import argparse
from typing import Dict, Any
from debugging.monitoring.health_checks import SystemHealthChecker, HealthStatus
from debugging.log_analysis.correlation import LogCorrelator
from debugging.profiling.performance_analyzer import PerformanceAnalyzer

def debug_pipeline_execution(pipeline_id: str, run_id: str, log_file_path: str):
    """Debug a specific pipeline execution using multiple analysis techniques."""

    print(f"Debugging pipeline {pipeline_id}, run {run_id}")
    print("=" * 60)

    # 1. System health check
    print("\n1. SYSTEM HEALTH CHECK")
    print("-" * 30)
    health_checker = SystemHealthChecker(
        db_connection_string="postgresql://user:pass@localhost/etl_db",
        message_broker_url="amqp://localhost:5672"
    )

    resource_health = health_checker.check_system_resources()
    db_health = health_checker.check_database_connectivity()

    print(f"System Resources: {resource_health.status.value}")
    print(f"  {resource_health.message}")
```

```
print(f"Database: {db_health.status.value}")
print(f"  {db_health.message}")

if resource_health.status == HealthStatus.UNHEALTHY:
    print("\n⚠️  CRITICAL: System resources are unhealthy. This may be causing pipeline issues.")

# 2. Log analysis
print("\n2. LOG ANALYSIS")
print("-" * 30)

correlator = LogCorrelator()
log_entries = []

# Parse log file
try:
    with open(log_file_path, 'r') as f:
        for line in f:
            entry = correlator.parse_log_entry(line)
            if entry:
                log_entries.append(entry)

    print(f"Parsed {len(log_entries)} log entries")

    # Find logs for this pipeline run
    run_logs = correlator.correlate_pipeline_logs(log_entries, run_id)

    print(f"Found logs from {len(run_logs)} components for run {run_id}")

    # Look for errors
    error_logs = [entry for entry in log_entries
                  if entry.run_id == run_id and entry.level == 'ERROR']

    if error_logs:
```

```
            print(f"\n⚠️  Found {len(error_logs)} error entries:")

            for error in error_logs:

                print(f"  {error.timestamp} [{error.component}] {error.message}")



                # Get context around each error

                context = correlator.find_error_context(log_entries, error, context_seconds=30)

                print(f"    Context ({len(context)} entries around error):")

                for ctx_entry in context[-5:]:  # Show last 5 context entries

                    print(f"      {ctx_entry.timestamp} [{ctx_entry.component}] {ctx_entry.level}:
{ctx_entry.message}")

        else:

            print("✓ No errors found in logs")



    except FileNotFoundError:

        print(f"⚠️  Log file not found: {log_file_path}")

    except Exception as e:

        print(f"⚠️  Error parsing logs: {e}")



    # 3. Performance analysis suggestions

    print("\n3. PERFORMANCE ANALYSIS SUGGESTIONS")

    print("-" * 30)



    print("To analyze performance issues:")

    print("1. Enable performance monitoring for the next pipeline run:")

    print(f"   analyzer = PerformanceAnalyzer()")

    print(f"   analyzer.start_monitoring('{run_id}')")

    print("   # Run pipeline")

    print(f"   results = analyzer.stop_monitoring()")

    print("\n2. Check database query performance:")

    print("   - Enable query logging in database configuration")

    print("   - Look for slow queries in database logs")

    print("   - Check for missing indexes on large tables")

    print("\n3. Monitor resource usage patterns:")
```

```
    print("   - Memory growth during transformation phases")
    print("   - CPU spikes during data processing")
    print("   - Network usage during extraction/loading")

def main():
    parser = argparse.ArgumentParser(description="Debug ETL pipeline execution")
    parser.add_argument("--pipeline-id", required=True, help="Pipeline ID to debug")
    parser.add_argument("--run-id", required=True, help="Pipeline run ID to debug")
    parser.add_argument("--log-file", required=True, help="Path to log file")

    args = parser.parse_args()

    debug_pipeline_execution(args.pipeline_id, args.run_id, args.log_file)

if __name__ == "__main__":
    main()
```

Milestone Checkpoints for Debugging Implementation

After implementing health checking:

- Run: `python -m debugging.monitoring.health_checks`
- Expected: Health status reports for system resources and database connectivity
- Verify: Health checks correctly identify resource pressure and connection issues

After implementing log correlation:

- Run: `python scripts/debug_pipeline.py --pipeline-id test-pipeline --run-id run-123 --log-file pipeline.log`
- Expected: Parsed log entries grouped by component with error context
- Verify: Error messages are correctly correlated with surrounding context

After implementing performance monitoring:

- Run a pipeline with performance monitoring enabled
- Expected: Real-time metrics collection during execution
- Verify: Memory and CPU usage patterns are captured and can identify bottlenecks

Common Debugging Issues and Solutions

Symptom	Likely Cause	Diagnosis Command	Fix
Debug script fails to connect to database	Connection string incorrect or database unavailable	`psql -d "postgresql://user:pass@localhost/etl_db" -c "SELECT 1"`	Update connection string, verify database is running
Log parsing returns empty results	Log format doesn't match expected structure	Check log file manually, verify timestamp format matches regex	Update log pattern regex or fix log format
Performance monitoring shows no data	Background thread not starting or crashing	Add exception handling and logging to monitoring loop	Fix threading issues, handle psutil exceptions
Health checks always report unhealthy	Thresholds too strict for environment	Adjust resource usage thresholds in health check logic	Tune thresholds based on system capacity

Future Extensions

> **Milestone(s):** All milestones - potential enhancements that build upon pipeline definition (Milestone 1), data processing capabilities (Milestones 2-3), and orchestration infrastructure (Milestone 4) to enable advanced use cases and cloud-scale deployment.

Mental Model: Growing City Infrastructure

Think of our ETL system like a small city that's planning for growth. Right now we've built the essential infrastructure - roads (data pipelines), traffic lights (orchestration), utilities (data processing), and a city hall (monitoring). But as the city grows, we'll need highways for high-speed traffic (distributed processing), airports for long-distance connections (cloud integration), smart traffic systems (real-time processing), and specialized districts (machine learning workflows). The key is designing today's infrastructure so it can evolve into tomorrow's metropolis without requiring a complete rebuild.

The current ETL system provides a solid foundation with its DAG-based pipeline definition, pluggable connector architecture, transformation engine, and orchestration framework. However, production environments often demand capabilities beyond traditional batch processing - from handling massive data volumes that require distributed computing, to processing streaming data in real-time, to supporting machine learning workflows with specialized requirements. This section explores how the existing architecture can be extended to support these advanced scenarios while maintaining the system's core principles of reliability, observability, and ease of use.

The extensions fall into two main categories: scalability improvements that help the system handle larger workloads and more complex deployment scenarios, and advanced pipeline features that enable new types of data processing workflows. Each extension is designed to build incrementally on the existing foundation, allowing teams to adopt only the capabilities they need while maintaining backward compatibility with existing pipelines.

Scalability Extensions

Distributed Task Execution

The current system executes tasks on a single machine using thread-based parallelization within each execution level of the DAG. While this approach works well for moderate workloads, large-scale data processing often requires distributing task execution across

multiple machines to handle datasets that exceed single-machine memory and CPU capacity, or to achieve processing speeds that require parallel computation.

> **Decision: Distributed Execution Architecture**
>
> - **Context**: Single-machine execution limits throughput and dataset size, preventing the system from handling enterprise-scale workloads that may involve terabytes of data or complex transformations requiring substantial computational resources.
> - **Options Considered**:
>   1. Container-based task execution with Kubernetes orchestration
>   2. Spark-based distributed processing integration
>   3. Custom distributed worker pool with message queues
> - **Decision**: Hybrid approach using containerized task execution for general tasks with optional Spark integration for data-intensive transformations
> - **Rationale**: Containers provide isolation and resource control for arbitrary tasks, while Spark integration leverages existing distributed processing expertise for heavy data workloads
> - **Consequences**: Enables horizontal scaling and large dataset processing, but requires container orchestration infrastructure and adds deployment complexity

The distributed execution model extends the existing `TaskExecution` framework by introducing execution targets that can be either local threads or remote execution environments. The `ExecutionPlan` component gains new capabilities to consider resource requirements and execution target availability when scheduling tasks across the cluster.

Component	Responsibility	Distribution Strategy
Distributed Task Scheduler	Assigns tasks to available execution targets based on resource requirements	Round-robin with resource consideration
Container Task Executor	Executes tasks in isolated containers with configurable resource limits	One container per task execution
Spark Integration Layer	Submits data-intensive transformations to Spark clusters	Spark job per transformation task
Resource Manager	Tracks available compute resources across execution targets	Periodic heartbeat with capacity reporting
Result Collector	Aggregates task results from distributed executions	Streaming collection with partial results

The distributed execution workflow follows these steps:

1. The `ExecutionPlan` analyzer examines each task's resource requirements and estimated data volume to determine optimal execution target
2. Tasks requiring large memory or CPU are marked for container execution, while data transformations exceeding size thresholds are routed to Spark
3. The Distributed Task Scheduler maintains a registry of available execution targets with their current resource utilization
4. When a task becomes ready for execution, the scheduler selects an appropriate target and packages the task with its dependencies
5. Container-based tasks are submitted to Kubernetes with resource limits and environment variables containing connection credentials
6. Spark-based tasks are converted to Spark jobs with optimized partitioning strategies based on input data characteristics
7. The Result Collector streams task outputs back to the central orchestrator and updates task state in the shared metadata store
8. Failed tasks can be rescheduled on different execution targets, with automatic retry logic adapted for distributed failure scenarios

Container task execution requires packaging each task's execution environment, including the necessary connector libraries, transformation functions, and configuration data. The system creates lightweight container images containing the ETL runtime and mounts task-specific configuration and credentials at execution time.

Container Component	Purpose	Configuration Source
ETL Runtime Base Image	Contains Python environment and core ETL libraries	Pre-built and versioned container registry
Task Configuration Mount	Task definition, parameters, and connection configs	ConfigMap generated from TaskDefinition
Credential Mount	Database passwords, API keys, and certificates	Kubernetes secrets with rotation support
Shared Storage Mount	Large intermediate datasets and checkpoint files	Persistent volume or object storage mount
Log Collection Sidecar	Streams task logs back to central monitoring	Fluent Bit or similar log shipping agent

Spark integration focuses on data transformations that benefit from distributed processing, particularly large joins, aggregations, and complex analytical workloads. The integration layer translates SQL-based transformations into Spark SQL jobs and provides a framework for registering Python UDFs as Spark user-defined functions.

Horizontal Scaling and Auto-scaling

The current orchestration system runs as a single process managing pipeline scheduling and execution coordination. Production deployments require horizontal scaling to handle increased pipeline throughput, support high availability during component failures, and automatically adjust capacity based on workload demand.

The horizontal scaling model introduces multiple orchestrator instances that coordinate through a shared metadata store and message broker. Each instance can handle pipeline scheduling, task execution coordination, and monitoring responsibilities, with automatic failover when instances become unavailable.

Scaling Component	Single Instance Role	Multi-Instance Coordination
Pipeline Scheduler	Triggers pipelines based on schedules	Leader election with schedule ownership
Task Execution Coordinator	Manages task state and dependencies	Distributed coordination via message queue
Monitoring Collector	Aggregates metrics and logs	Partition-based collection with merge
Metadata Store Access	Direct database connections	Connection pooling with read replicas
Message Broker Client	Simple pub/sub for events	Consumer groups with partition assignment

Leader election ensures that only one orchestrator instance schedules each pipeline to prevent duplicate executions, while multiple instances can coordinate task execution within the same pipeline run. The system uses etcd or a similar consensus system for leader election, with lease renewal and automatic failover when the current leader becomes unavailable.

Auto-scaling responds to increased workload by monitoring key metrics and adjusting the number of orchestrator instances and execution targets. The auto-scaling controller tracks pipeline queue length, average task execution time, and resource utilization to make scaling decisions.

Scaling Trigger	Scale Up Condition	Scale Down Condition	Scaling Action
Pipeline Queue Length	>50 queued pipelines	<10 queued pipelines	Add/remove orchestrator instances
Task Execution Wait Time	>5 minutes average wait	<1 minute average wait	Add/remove execution workers
Resource Utilization	>80% CPU/memory usage	<30% CPU/memory usage	Adjust container resource limits
Error Rate Spike	>10% task failure rate	Normal failure rates	Temporarily reduce concurrency

The auto-scaling system includes safeguards to prevent rapid scaling oscillation and considers the cost implications of adding resources. Scale-down decisions include grace periods to allow in-progress tasks to complete and evaluate recent scaling actions to avoid immediate reversals.

> **Critical Design Insight**: Auto-scaling decisions must account for the stateful nature of ETL pipelines, where scaling down may interrupt long-running transformations or cause data inconsistency if not coordinated properly with task execution state.

Cloud-Native Integration

Modern ETL deployments increasingly leverage cloud services for storage, compute, and managed infrastructure services. The cloud-native extensions integrate with cloud provider APIs to dynamically provision resources, leverage managed services for data storage and processing, and implement cloud-specific optimization patterns.

The cloud integration layer provides abstractions over common cloud services while maintaining the ability to deploy in on-premises or hybrid environments. The system detects its deployment environment and automatically configures appropriate service integrations.

Cloud Service Category	AWS Integration	GCP Integration	Azure Integration
Object Storage	S3 with IAM roles	Cloud Storage with service accounts	Blob Storage with managed identity
Managed Databases	RDS, Redshift connections	Cloud SQL, BigQuery connectors	SQL Database, Synapse Analytics
Container Orchestration	EKS with Fargate support	GKE with Autopilot	AKS with virtual nodes
Serverless Compute	Lambda for lightweight tasks	Cloud Functions integration	Azure Functions support
Message Queues	SQS/SNS for coordination	Pub/Sub for event streaming	Service Bus for messaging
Monitoring Integration	CloudWatch metrics/logs	Cloud Monitoring/Logging	Azure Monitor integration

Cloud-native deployment leverages Infrastructure as Code (IaC) tools to provision and configure the required cloud resources. The system includes Terraform modules and Kubernetes Helm charts that can deploy the complete ETL infrastructure with appropriate security configurations, network policies, and monitoring integrations.

Serverless integration allows lightweight tasks to run in cloud functions rather than requiring persistent compute resources. Tasks suitable for serverless execution include data validation, simple transformations, and notification delivery. The system automatically identifies serverless-compatible tasks based on resource requirements and execution patterns.

Serverless Criteria	Eligible Task Types	Execution Environment	Limitations
Memory < 1GB	Validation, notification tasks	Lambda, Cloud Functions	15-minute maximum execution time
No persistent state	Stateless transformations	Function runtime with mounted storage	No local file system persistence
Predictable runtime	Simple data operations	Auto-scaling function instances	Cold start latency considerations
Standard dependencies	Tasks using built-in libraries	Pre-packaged runtime environment	Limited custom library support

Cloud storage integration optimizes data transfer by leveraging cloud-native features like transfer acceleration, regional storage, and intelligent tiering. Large datasets are processed using cloud-specific optimization patterns, such as S3 Transfer Acceleration for fast uploads and BigQuery's columnar storage for analytical workloads.

⚠️ **Pitfall: Cloud Vendor Lock-in** Directly using cloud-specific APIs throughout the codebase creates tight coupling that makes migration difficult. The abstraction layer must provide cloud-agnostic interfaces while still allowing access to cloud-specific optimizations when needed. This requires careful interface design that balances portability with performance.

Advanced Pipeline Features

Stream Processing Integration

Traditional ETL systems focus on batch processing, where data is processed in discrete chunks at scheduled intervals. However, many modern use cases require real-time or near-real-time processing of continuously arriving data streams. Stream processing integration extends the DAG-based pipeline model to support continuous data flows while maintaining the existing batch processing capabilities.

> **Decision: Hybrid Stream-Batch Processing Model**
>
> - **Context**: Organizations need both batch processing for historical data and stream processing for real-time analytics, requiring a unified system that can handle both paradigms without forcing users to maintain separate infrastructures.
> - **Options Considered**:
>   1. Pure stream processing with micro-batching for historical data
>   2. Separate stream and batch systems with shared metadata
>   3. Unified DAG model supporting both stream and batch tasks
> - **Decision**: Extended DAG model with stream-aware task types and unified orchestration
> - **Rationale**: Maintains familiar DAG abstraction while enabling stream processing, allows gradual migration from batch to stream, and provides unified monitoring and debugging experience
> - **Consequences**: Enables real-time use cases with consistent tooling, but adds complexity to scheduling and state management for continuous processing

The stream processing model introduces new task types that operate on continuous data streams rather than discrete datasets. Stream tasks maintain persistent execution state and process data as it arrives, while batch tasks continue to operate on complete datasets at scheduled intervals.

Task Type	Data Model	Execution Pattern	State Management
Stream Source	Continuous event stream	Always running with checkpoint recovery	Maintains consumer offset/position
Stream Transform	Input stream → Output stream	Stateful processing with windowing	Checkpoint-based state persistence
Stream Sink	Stream → External system	Micro-batch writes with exactly-once semantics	Idempotent write tracking
Batch Task	Complete dataset	Scheduled execution	Traditional task state machine
Hybrid Task	Stream + Batch inputs	Triggered on schedule or stream events	Mixed state with clear boundaries

Stream processing requires modified dependency semantics where stream tasks can have dependencies on both other stream tasks (for streaming transformations) and batch tasks (for enrichment data). The DAG validation extends to verify that stream dependencies form valid topologies and that batch dependencies provide appropriate data freshness guarantees.

The stream processing workflow follows these principles:

1. Stream source tasks connect to message queues, event streams, or change data capture systems and continuously consume new events
2. Each consumed event or micro-batch triggers downstream stream transformations following the DAG dependency structure
3. Stream transforms maintain processing state (windows, aggregations, join state) with regular checkpointing to enable failure recovery
4. Stream sinks buffer output events and write to destinations using configurable batching strategies for efficiency
5. Hybrid tasks combine stream data with batch-processed reference data, triggering reprocessing when either input changes
6. The orchestrator monitors stream task health and restarts failed stream tasks from the last successful checkpoint

7. Stream processing metrics track throughput, latency, and backlog to enable monitoring and auto-scaling decisions

Stream task checkpointing ensures exactly-once processing semantics by atomically saving processing state along with output records. The checkpoint mechanism coordinates with downstream tasks to maintain consistency across the entire stream processing pipeline.

Checkpoint Component	Purpose	Persistence Strategy
Consumer Offset	Tracks position in input stream	Stored in message broker or external store
Processing State	Window contents, aggregation values	Serialized to persistent storage with versioning
Output Tracking	Records successfully written downstream	Deduplicated output log with retention policy
Watermark Position	Event time progress for windowed operations	Coordinated across parallel processing instances

Machine Learning Pipeline Integration

Data pipelines increasingly serve machine learning workloads that have specialized requirements around model training, feature engineering, model deployment, and inference serving. The ML integration extends the transformation engine to support ML-specific operations while leveraging existing orchestration and monitoring infrastructure.

Machine learning workflows typically involve feature extraction from raw data, model training on prepared datasets, model validation and testing, and deployment to serving infrastructure. These workflows have unique characteristics including long-running training jobs, iterative experimentation, model versioning, and performance monitoring that differ from traditional ETL operations.

The ML pipeline extension introduces specialized task types that integrate with popular ML frameworks and provide abstractions for common ML operations. These tasks can be combined with traditional ETL tasks in the same DAG to create end-to-end ML pipelines.

ML Task Type	Purpose	Framework Integration	Output Artifacts
Feature Engineering	Transform raw data to ML features	Pandas, Spark MLlib	Feature datasets with schema
Model Training	Train ML models on prepared data	Scikit-learn, TensorFlow, PyTorch	Versioned model artifacts
Model Evaluation	Validate model performance	MLflow, Weights & Biases	Performance metrics and reports
Model Deployment	Deploy models to serving infrastructure	KubeFlow, SageMaker, MLflow	Deployed model endpoints
Batch Inference	Score large datasets with trained models	Spark, Dask for distributed scoring	Scored datasets with predictions
Model Monitoring	Track model performance in production	Custom monitoring with alerting	Drift detection and performance metrics

Feature engineering tasks extend the transformation engine with ML-specific operations like feature scaling, encoding categorical variables, handling missing values, and creating time-based features. These transformations maintain lineage tracking to enable feature attribution and debugging model performance issues.

Model training tasks handle the specialized requirements of ML training including experiment tracking, hyperparameter optimization, cross-validation, and distributed training for large models. The training integration provides abstractions that work across different ML frameworks while maintaining flexibility for framework-specific optimizations.

Training Feature	Implementation	Framework Support
Experiment Tracking	MLflow integration for logging	All major frameworks via MLflow
Hyperparameter Tuning	Optuna-based optimization	Framework-agnostic optimization
Distributed Training	Multi-GPU and multi-node support	TensorFlow distributed, PyTorch DDP
Model Versioning	Git-like versioning for model artifacts	MLflow Model Registry integration
Resource Management	GPU allocation and scheduling	Kubernetes resource quotas

Model deployment tasks automate the process of taking trained models and making them available for inference, whether through batch scoring jobs or real-time serving endpoints. The deployment integration handles model packaging, dependency management, and infrastructure provisioning.

> **Critical Design Insight**: ML pipelines often require iterative development where data scientists experiment with different feature engineering and modeling approaches. The pipeline system must support branching and merging of experimental workflows while maintaining reproducibility and version control of successful experiments.

The ML integration includes specialized monitoring for deployed models that tracks prediction accuracy, data drift, and performance degradation over time. This monitoring can trigger automatic retraining pipelines when model performance drops below acceptable thresholds.

Real-time Processing and Event-driven Orchestration

Beyond stream processing for continuous data flows, real-time processing enables sub-second response times for critical business processes and event-driven orchestration that responds immediately to external events rather than relying solely on schedule-based triggers.

Real-time processing requirements include low-latency data transformation, immediate alerting on anomaly detection, real-time feature serving for ML models, and rapid response to business events. These use cases require processing architectures optimized for latency rather than throughput, with careful attention to resource allocation and prioritization.

Event-driven orchestration extends the scheduling system to trigger pipelines based on external events such as file arrivals, database changes, API calls, or message queue events. This enables reactive data processing that responds to business events as they occur rather than waiting for the next scheduled batch window.

Event Source	Trigger Mechanism	Latency Target	Use Cases
File System Events	inotify, S3 events	<1 second	Process files immediately upon arrival
Database Changes	CDC, triggers	<5 seconds	Sync data changes across systems
API Webhooks	HTTP endpoints	<500ms	React to external system notifications
Message Queues	Real-time consumers	<100ms	Process high-priority business events
Monitoring Alerts	Alert manager integration	<30 seconds	Trigger remediation pipelines

Real-time processing tasks use in-memory data structures and optimized execution paths to minimize latency. The system provides priority queues for real-time tasks and resource reservation to ensure adequate capacity for time-sensitive processing.

The event-driven architecture includes event routing and filtering capabilities that allow pipelines to subscribe to specific types of events and apply filters to process only relevant events. This prevents overwhelming the system with low-priority events while ensuring critical events receive immediate attention.

Event Processing Stage	Purpose	Performance Optimization
Event Ingestion	Receive and queue incoming events	High-throughput async I/O
Event Filtering	Apply subscription and filter rules	In-memory rule evaluation
Event Routing	Direct events to appropriate pipelines	Hash-based partition assignment
Priority Scheduling	Schedule real-time tasks with priority	Dedicated resource pools
Low-latency Execution	Execute time-sensitive transformations	Pre-warmed execution environments

Real-time pipeline orchestration includes circuit breakers and bulkhead patterns to prevent cascading failures when external systems become slow or unavailable. The system can automatically route processing to backup systems or degrade functionality gracefully when real-time processing targets cannot be met.

⚠️ **Pitfall: Real-time Complexity Trade-offs** Optimizing for real-time processing often conflicts with other system qualities like reliability, consistency, and cost-effectiveness. Real-time capabilities should be applied selectively to use cases that truly require low latency, while maintaining robust batch processing for operations where eventual consistency is acceptable. Over-engineering for real-time requirements can significantly increase system complexity and operational overhead.

Implementation Guidance

Technology Recommendations

Extension Category	Simple Option	Advanced Option
Container Orchestration	Docker Compose for development	Kubernetes with Helm charts
Distributed Computing	Celery with Redis backend	Apache Spark on Kubernetes
Message Broker	Redis Pub/Sub	Apache Kafka with Schema Registry
Stream Processing	Python asyncio with Redis Streams	Apache Kafka Streams or Apache Flink
Cloud Integration	boto3 for AWS, basic cloud APIs	Terraform + cloud SDKs with IAM roles
ML Framework	Scikit-learn with joblib	MLflow + Kubeflow for enterprise ML
Real-time Processing	WebSockets + in-memory queues	Apache Pulsar with function computing
Auto-scaling	Simple threshold-based scaling	Kubernetes HPA with custom metrics

Recommended File Structure Extension

```
project-root/
  etl_system/
    extensions/
      distributed/
        __init__.py
        container_executor.py      ← Container-based task execution
        spark_integration.py       ← Spark job submission and monitoring
        resource_manager.py        ← Cluster resource tracking
        distributed_scheduler.py   ← Multi-target task scheduling

      scaling/
        __init__.py
        auto_scaler.py            ← Auto-scaling controller
        leader_election.py        ← Multi-instance coordination
        load_balancer.py          ← Request distribution

      cloud/
        __init__.py
        aws_integration.py        ← AWS service connectors
        gcp_integration.py        ← Google Cloud service connectors
        azure_integration.py      ← Azure service connectors
        cloud_storage.py          ← Multi-cloud storage abstraction

      streaming/
        __init__.py
        stream_tasks.py           ← Stream processing task types
        checkpoint_manager.py     ← Stream state checkpointing
        event_router.py          ← Event-driven pipeline triggers

      ml/
        __init__.py
        feature_engineering.py    ← ML feature transformation tasks
        model_training.py         ← ML training task types
        model_serving.py          ← Model deployment and serving
        experiment_tracking.py    ← MLflow integration

      realtime/
        __init__.py
        priority_scheduler.py     ← Low-latency task scheduling
        event_processor.py        ← Real-time event processing
        circuit_breaker.py        ← Failure protection patterns

    config/
      extensions/
        distributed.yaml          ← Container and Spark configuration
        cloud.yaml               ← Cloud provider settings
        streaming.yaml           ← Stream processing configuration
        ml.yaml                  ← ML framework settings
```

Infrastructure Starter Code

Container Task Executor Implementation:

PYTHON

```python
import asyncio

import logging

from typing import Dict, Any, Optional

import docker

import kubernetes

from kubernetes import client, config

from dataclasses import dataclass


@dataclass

class ContainerTaskSpec:

    """Specification for containerized task execution."""

    task_id: str

    image: str

    command: list

    environment: Dict[str, str]

    resource_limits: Dict[str, str]

    volumes: Dict[str, str]

    timeout_seconds: int


class ContainerTaskExecutor:

    """Executes ETL tasks in isolated containers using Kubernetes."""



    def __init__(self, namespace: str = "etl-system"):

        self.namespace = namespace

        self.k8s_client = None

        self.logger = logging.getLogger(__name__)



        try:

            config.load_incluster_config()  # Running in cluster

        except kubernetes.config.ConfigException:

            config.load_kube_config()  # Local development



        self.k8s_client = client.BatchV1Api()

        self.core_client = client.CoreV1Api()
```

```python
async def execute_task(self, task_spec: ContainerTaskSpec) -> Dict[str, Any]:
    """
    Execute a task in a Kubernetes job container.

    Returns:
        Dict containing execution results, logs, and metadata
    """
    job_name = f"etl-task-{task_spec.task_id}"

    try:
        # Create Kubernetes job specification
        job_spec = self._create_job_spec(job_name, task_spec)

        # Submit job to Kubernetes
        self.k8s_client.create_namespaced_job(
            namespace=self.namespace,
            body=job_spec
        )

        # Wait for completion and collect results
        result = await self._wait_for_completion(job_name, task_spec.timeout_seconds)

        # Cleanup completed job
        await self._cleanup_job(job_name)

        return result

    except Exception as e:
        self.logger.error(f"Container task execution failed: {e}")
        await self._cleanup_job(job_name)
        raise
```

```
    def _create_job_spec(self, job_name: str, task_spec: ContainerTaskSpec):
        """Create Kubernetes Job specification for ETL task."""
        return client.V1Job(
            metadata=client.V1ObjectMeta(name=job_name),
            spec=client.V1JobSpec(
                template=client.V1PodTemplateSpec(
                    metadata=client.V1ObjectMeta(labels={"app": "etl-task"}),
                    spec=client.V1PodSpec(
                        restart_policy="Never",
                        containers=[
                            client.V1Container(
                                name="etl-task",
                                image=task_spec.image,
                                command=task_spec.command,
                                env=[
                                    client.V1EnvVar(name=k, value=v)
                                    for k, v in task_spec.environment.items()
                                ],
                                resources=client.V1ResourceRequirements(
                                    limits=task_spec.resource_limits
                                ),
                                volume_mounts=[
                                    client.V1VolumeMount(
                                        name=name,
                                        mount_path=path
                                    )
                                    for name, path in task_spec.volumes.items()
                                ]
                            )
                        ],
                        volumes=[
                            client.V1Volume(
                                name=name,
```

```
                            persistent_volume_claim=client.V1PersistentVolumeClaimVolumeSource(
                                claim_name=f"{name}-pvc"
                            )
                        )
                        for name in task_spec.volumes.keys()
                    ]
                )
            )
        )
    )

async def _wait_for_completion(self, job_name: str, timeout_seconds: int) -> Dict[str, Any]:
    """Wait for job completion and return results."""
    start_time = asyncio.get_event_loop().time()

    while (asyncio.get_event_loop().time() - start_time) < timeout_seconds:
        # Check job status
        job = self.k8s_client.read_namespaced_job_status(
            name=job_name,
            namespace=self.namespace
        )

        if job.status.succeeded:
            # Collect logs and return success result
            logs = await self._collect_logs(job_name)
            return {
                "status": "success",
                "logs": logs,
                "duration": asyncio.get_event_loop().time() - start_time
            }

        elif job.status.failed:
            # Collect logs and return failure result
```

```
            logs = await self._collect_logs(job_name)
            return {
                "status": "failed",
                "logs": logs,
                "error": "Container job failed"
            }

        # Wait before checking again
        await asyncio.sleep(5)

    # Timeout occurred
    return {
        "status": "timeout",
        "error": f"Task exceeded {timeout_seconds} second timeout"
    }

async def _collect_logs(self, job_name: str) -> list:
    """Collect logs from completed job pods."""
    try:
        # Find pods created by the job
        pods = self.core_client.list_namespaced_pod(
            namespace=self.namespace,
            label_selector=f"job-name={job_name}"
        )

        logs = []
        for pod in pods.items:
            pod_logs = self.core_client.read_namespaced_pod_log(
                name=pod.metadata.name,
                namespace=self.namespace
            )
            logs.append(pod_logs)
```

```
            return logs


        except Exception as e:
            self.logger.error(f"Failed to collect logs: {e}")
            return [f"Log collection failed: {e}"]


    async def _cleanup_job(self, job_name: str):
        """Clean up completed Kubernetes job and associated pods."""
        try:
            # Delete the job (this also deletes associated pods)
            self.k8s_client.delete_namespaced_job(
                name=job_name,
                namespace=self.namespace,
                propagation_policy="Foreground"
            )
        except Exception as e:
            self.logger.error(f"Failed to cleanup job {job_name}: {e}")
```

Stream Processing Task Framework:

```python
import asyncio

import json

import logging

from abc import ABC, abstractmethod

from typing import Dict, Any, AsyncIterator, Optional

from dataclasses import dataclass

from datetime import datetime


@dataclass

class StreamCheckpoint:

    """Stream processing checkpoint for recovery."""

    task_id: str

    stream_position: str

    processing_state: Dict[str, Any]

    timestamp: datetime


class StreamTask(ABC):

    """Abstract base class for stream processing tasks."""



    def __init__(self, task_id: str, config: Dict[str, Any]):

        self.task_id = task_id

        self.config = config

        self.logger = logging.getLogger(f"{__name__}.{task_id}")

        self.checkpoint_manager = None

        self.is_running = False



    @abstractmethod

    async def process_event(self, event: Dict[str, Any]) -> Optional[Dict[str, Any]]:

        """

        Process a single event from the stream.



        Args:

            event: Input event data
```

```
        Returns:
            Transformed event or None to filter out
        """
        pass

    @abstractmethod
    async def create_checkpoint(self) -> StreamCheckpoint:
        """Create checkpoint for current processing state."""
        pass

    @abstractmethod
    async def restore_from_checkpoint(self, checkpoint: StreamCheckpoint):
        """Restore processing state from checkpoint."""
        pass

    async def run(self, input_stream: AsyncIterator[Dict[str, Any]],
                  output_callback: callable):
        """Main execution loop for stream processing."""
        self.is_running = True
        checkpoint_counter = 0

        try:
            async for event in input_stream:
                if not self.is_running:
                    break

                # Process the event
                result = await self.process_event(event)

                if result is not None:
                    await output_callback(result)

                # Checkpoint periodically
```

```
                checkpoint_counter += 1
                if checkpoint_counter >= self.config.get("checkpoint_interval", 1000):
                    await self._create_checkpoint()
                    checkpoint_counter = 0

        except Exception as e:
            self.logger.error(f"Stream processing failed: {e}")
            raise
        finally:
            self.is_running = False

    async def stop(self):
        """Gracefully stop stream processing."""
        self.is_running = False

    async def _create_checkpoint(self):
        """Internal checkpoint creation with error handling."""
        try:
            if self.checkpoint_manager:
                checkpoint = await self.create_checkpoint()
                await self.checkpoint_manager.save_checkpoint(checkpoint)
        except Exception as e:
            self.logger.error(f"Checkpoint creation failed: {e}")

class WindowedAggregationTask(StreamTask):
    """Stream task that performs windowed aggregations."""

    def __init__(self, task_id: str, config: Dict[str, Any]):
        super().__init__(task_id, config)
        self.window_size = config.get("window_size_seconds", 60)
        self.aggregation_func = config.get("aggregation", "sum")
        self.group_by_field = config.get("group_by", "key")
```

```
        # Processing state
        self.current_window = {}
        self.window_start = None

    async def process_event(self, event: Dict[str, Any]) -> Optional[Dict[str, Any]]:
        """Aggregate events within time windows."""
        event_time = datetime.fromisoformat(event.get("timestamp"))

        # Initialize window if needed
        if self.window_start is None:
            self.window_start = event_time

        # Check if event belongs to current window
        if (event_time - self.window_start).total_seconds() <= self.window_size:
            # Add to current window
            group_key = event.get(self.group_by_field, "default")
            current_value = self.current_window.get(group_key, 0)

            if self.aggregation_func == "sum":
                self.current_window[group_key] = current_value + event.get("value", 0)
            elif self.aggregation_func == "count":
                self.current_window[group_key] = current_value + 1

            return None  # No output until window closes
        else:
            # Window is complete, emit results and start new window
            result = {
                "window_start": self.window_start.isoformat(),
                "window_end": (self.window_start + timedelta(seconds=self.window_size)).isoformat(),
                "aggregations": self.current_window.copy()
            }

            # Start new window
```

```
            self.current_window = {event.get(self.group_by_field, "default"): event.get("value", 0)}
            self.window_start = event_time


            return result


    async def create_checkpoint(self) -> StreamCheckpoint:
        """Save current window state for recovery."""
        return StreamCheckpoint(
            task_id=self.task_id,
            stream_position="",  # Would be set by stream consumer
            processing_state={
                "current_window": self.current_window,
                "window_start": self.window_start.isoformat() if self.window_start else None
            },
            timestamp=datetime.now()
        )


    async def restore_from_checkpoint(self, checkpoint: StreamCheckpoint):
        """Restore window state from checkpoint."""
        state = checkpoint.processing_state
        self.current_window = state.get("current_window", {})
        window_start_str = state.get("window_start")
        self.window_start = datetime.fromisoformat(window_start_str) if window_start_str else None
```

Core Logic Skeleton Code

Distributed Execution Coordinator:

PYTHON

```python
from typing import Dict, List, Optional
from dataclasses import dataclass
from enum import Enum

class ExecutionTarget(Enum):
    LOCAL = "local"
    CONTAINER = "container"
    SPARK = "spark"
    SERVERLESS = "serverless"

@dataclass
class ResourceRequirements:
    cpu_cores: float
    memory_gb: float
    gpu_count: int = 0
    storage_gb: float = 0


@dataclass
class ExecutionTargetInfo:
    target_id: str
    target_type: ExecutionTarget
    available_resources: ResourceRequirements
    current_utilization: ResourceRequirements

class DistributedExecutionCoordinator:
    """Coordinates task execution across multiple execution targets."""


    def __init__(self):
        self.execution_targets: Dict[str, ExecutionTargetInfo] = {}
        self.task_assignments: Dict[str, str] = {}  # task_id -> target_id


    def register_execution_target(self, target_info: ExecutionTargetInfo):
        """Register new execution target with available resources."""
        # TODO 1: Add target to execution_targets registry
```

```
        # TODO 2: Validate target configuration and connectivity
        # TODO 3: Start health monitoring for the target
        # TODO 4: Log target registration for debugging
        pass

    def select_execution_target(self, task_id: str, requirements: ResourceRequirements) -> Optional[str]:
        """
        Select optimal execution target for task based on resource requirements.

        Returns target_id of selected target, or None if no suitable target available.
        """
        # TODO 1: Filter targets that have sufficient available resources
        # TODO 2: Calculate suitability score based on current utilization
        # TODO 3: Consider target type preferences (container vs spark vs serverless)
        # TODO 4: Apply load balancing to distribute tasks evenly
        # TODO 5: Update target utilization after assignment
        # Hint: Score targets based on available_resources - current_utilization
        pass

    def estimate_task_resources(self, task_definition: 'TaskDefinition') -> ResourceRequirements:
        """Estimate resource requirements based on task configuration."""
        # TODO 1: Check task type and configuration for resource hints
        # TODO 2: Look up historical resource usage for similar tasks
        # TODO 3: Apply default resource requirements based on task type
        # TODO 4: Consider data volume estimates for transformation tasks
        # TODO 5: Add safety margins to prevent resource exhaustion
        pass

    def determine_execution_target_type(self, task_definition: 'TaskDefinition',
                                        requirements: ResourceRequirements) -> ExecutionTarget:
        """Determine appropriate execution target type for task."""
        # TODO 1: Check if task is suitable for serverless (stateless, short duration)
        # TODO 2: For large data transformations, prefer Spark execution
```

```
        # TODO 3: For I/O intensive tasks, prefer container execution
        # TODO 4: Fall back to local execution for simple tasks
        # TODO 5: Consider cost implications of different execution types
        pass
```

Auto-scaling Controller:

PYTHON

```python
from dataclasses import dataclass
from datetime import datetime, timedelta
import asyncio

@dataclass
class ScalingMetrics:
    pipeline_queue_length: int
    average_task_wait_time: float
    resource_utilization: float
    error_rate: float
    timestamp: datetime

@dataclass
class ScalingDecision:
    action: str  # "scale_up", "scale_down", "no_action"
    target_instances: int
    reason: str


class AutoScalingController:
    """Automatically scales ETL system resources based on workload metrics."""


    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.scaling_history: List[ScalingDecision] = []
        self.current_instances = 1
        self.last_scaling_time = None


    async def collect_metrics(self) -> ScalingMetrics:
        """Collect current system metrics for scaling decisions."""
        # TODO 1: Query pipeline scheduler for queue length
        # TODO 2: Calculate average task wait time from recent executions
        # TODO 3: Get resource utilization from all execution targets
        # TODO 4: Calculate error rate from recent task executions
        # TODO 5: Return ScalingMetrics with current timestamp
```

```
        pass

    def should_scale_up(self, metrics: ScalingMetrics) -> bool:
        """Determine if system should scale up based on metrics."""
        # TODO 1: Check if queue length exceeds scale-up threshold
        # TODO 2: Check if task wait time exceeds acceptable limits
        # TODO 3: Check if resource utilization is too high
        # TODO 4: Ensure cooldown period has elapsed since last scaling
        # TODO 5: Verify maximum instance limit not exceeded
        pass

    def should_scale_down(self, metrics: ScalingMetrics) -> bool:
        """Determine if system should scale down based on metrics."""
        # TODO 1: Check if queue length is below scale-down threshold
        # TODO 2: Check if resource utilization is low for sustained period
        # TODO 3: Ensure minimum instance count maintained
        # TODO 4: Verify no recent scaling actions (prevent oscillation)
        # TODO 5: Check that error rate is normal (don't scale down during issues)
        pass

    async def execute_scaling_decision(self, decision: ScalingDecision):
        """Execute the scaling decision by adjusting system resources."""
        # TODO 1: Log scaling decision with detailed reasoning
        # TODO 2: Update container orchestration (Kubernetes HPA or similar)
        # TODO 3: Wait for new instances to become ready
        # TODO 4: Verify scaling completed successfully
        # TODO 5: Update internal state and record scaling history
        # TODO 6: Send notification about scaling action
        pass
```

Milestone Checkpoints

Stream Processing Milestone: After implementing basic stream processing capabilities, verify:

1. **Stream Task Execution**: Create a simple windowed aggregation task that sums events over 30-second windows
    - Expected: Task should accumulate events and emit window results

- Command: `python -m etl_system.extensions.streaming.test_stream_task`
- Verify: Check that window boundaries are respected and aggregations are correct

2. **Checkpoint Recovery**: Stop and restart a stream task to verify checkpoint recovery
   - Expected: Task resumes from last checkpoint without data loss
   - Manual test: Send events, stop task mid-window, restart, verify window completion
   - Signs of issues: Duplicate events, lost aggregation state, incorrect window boundaries
3. **Event-driven Pipeline Triggers**: Set up file arrival event that triggers a pipeline
   - Expected: Pipeline starts within 1 second of file creation
   - Test: `touch /tmp/test_file.csv` should trigger pipeline execution
   - Verify: Check pipeline run logs show event-triggered execution

Distributed Execution Milestone: After implementing container-based execution:

1. **Container Task Execution**: Submit a simple data extraction task to Kubernetes
   - Expected: Task runs in container and returns results to orchestrator
   - Command: `kubectl logs -l app=etl-task` should show task execution logs
   - Verify: Task completion reported in pipeline run status
2. **Resource-based Target Selection**: Run tasks with different resource requirements
   - Expected: High-memory tasks assigned to appropriate execution targets
   - Test: Submit both light and heavy tasks, verify assignment logic
   - Signs of issues: Tasks assigned to under-resourced targets, execution failures
3. **Auto-scaling Response**: Generate high pipeline load to trigger scaling
   - Expected: Additional orchestrator instances start automatically
   - Monitor: `kubectl get pods` should show new instances after sustained load
   - Verify: Load distributes across instances, no duplicate pipeline executions

Debugging Tips

Symptom	Likely Cause	Diagnosis Steps	Fix
Container tasks never complete	Resource limits too restrictive	Check `kubectl describe pod` for resource constraints	Increase memory/CPU limits in container spec
Stream processing falls behind	Processing slower than event arrival rate	Monitor event queue length and processing latency	Add parallel stream processors or optimize processing logic
Auto-scaling oscillation	Thresholds too sensitive or cooldown too short	Review scaling history and metric patterns	Increase cooldown period and add hysteresis to thresholds
Distributed task failures	Network connectivity or authentication issues	Test connectivity from containers to data sources	Update network policies and credential mounting
ML pipeline memory errors	Model training exceeds available memory	Monitor memory usage during training	Use distributed training or gradient accumulation
Real-time processing timeouts	Processing chain too complex for latency target	Profile each processing step latency	Simplify processing or use faster execution targets
Cloud integration auth failures	IAM roles or service account misconfig	Check cloud provider logs and permissions	Update role policies and credential configuration
Stream checkpoint corruption	Concurrent checkpoint writes or storage failures	Check checkpoint storage logs and file integrity	Implement checkpoint locking and validation

Glossary

> **Milestone(s):** All milestones - comprehensive vocabulary reference that supports understanding across pipeline definition (Milestone 1), data processing (Milestones 2-3), orchestration and monitoring (Milestone 4), and system operations.

This glossary provides comprehensive definitions of key technical terms and domain-specific vocabulary used throughout the ETL system design document. The terms are organized to support both newcomers learning ETL concepts and experienced developers working with the system's specific implementation details.

Mental Model: Technical Dictionary with Context

Think of this glossary as a specialized technical dictionary that goes beyond simple definitions. Like a good dictionary for a foreign language, each entry provides not just the meaning but also context about when and how the term is used in ETL systems. Just as language dictionaries show pronunciation, etymology, and usage examples, this technical glossary explains relationships between concepts, common usage patterns, and potential pitfalls that arise when working with these terms.

The glossary serves as both a reference during development and a learning tool for understanding the broader ETL ecosystem. Terms build upon each other - understanding **watermarking** requires knowledge of **incremental loading**, which relates to **change data capture**, which connects to **idempotent** operations.

Core ETL and Data Processing Terms

Term	Definition	Context and Usage
ETL	Extract Transform Load - the three-phase process of moving data from sources to destinations with transformations applied	The fundamental pattern for data pipeline systems. Extract pulls data from sources, Transform applies business logic and data quality rules, Load writes results to destinations
DAG	Directed Acyclic Graph - a graph structure with directed edges and no cycles, used to represent task dependencies	Core data structure for pipeline definition. Tasks are nodes, dependencies are edges. Acyclic property ensures execution order can be determined through topological sorting
idempotent	Operations that produce the same result when executed multiple times, regardless of how many times they run	Critical property for ETL reliability. Enables safe retries after failures without data corruption or duplication. Example: `INSERT ... ON CONFLICT DO NOTHING`
topological sort	Algorithm for ordering nodes in a DAG such that all dependencies come before their dependents	Used to determine task execution order. Kahn's algorithm produces parallel execution levels, allowing multiple independent tasks to run simultaneously
checkpoint	Saving intermediate processing state to enable resumption after failures	Enables fault tolerance by avoiding complete restart after partial completion. Includes task state, processed record counts, and watermark positions
lineage	Tracking data provenance and transformation history through pipeline execution	Essential for compliance, debugging, and impact analysis. Records which source data contributed to each output record and what transformations were applied
watermark	High-water mark indicating the latest data that has been successfully processed	Key mechanism for incremental loading. Typically a timestamp, sequence number, or other monotonically increasing value that tracks processing progress
adjacency list	Graph representation that maps each node to a list of its direct neighbors	Standard representation for DAGs in pipeline systems. Enables efficient cycle detection and topological sorting algorithms
cycle detection	Algorithm to identify circular dependencies in directed graphs	Critical validation step before pipeline execution. Uses depth-first search with three-color marking (white/gray/black) to detect back edges
critical path	Longest dependency chain in a DAG, determining minimum possible execution time	Used for execution planning and performance optimization. Tasks on the critical path cannot be delayed without extending total pipeline runtime
execution levels	Groups of tasks that have no dependencies between them and can execute in parallel	Result of topological sorting. Level 0 contains tasks with no dependencies, Level N contains tasks whose dependencies are all in levels 0 through N-1
in-degree	Number of incoming dependencies for a task in the dependency graph	Used in Kahn's algorithm for topological sorting. Tasks with in-degree 0 are ready to execute

Data Extraction and Loading Terms

Term	Definition	Context and Usage
watermarking	Process of tracking and updating high-water marks for incremental data extraction	Prevents duplicate processing and enables efficient incremental loads. Watermarks must be updated atomically with successful data loading
cursor-based pagination	Using opaque tokens or identifiers to track position in paginated API results	More reliable than offset-based pagination for changing datasets. Cursors remain stable even when underlying data is modified during extraction
change data capture	Real-time stream of database changes (inserts, updates, deletes) for incremental loading	Enables near real-time data synchronization. Common implementations include database transaction logs, triggers, or timestamp-based change tracking
bulk loading	Optimized batch insertion technique designed for high throughput data transfer	Significantly faster than row-by-row inserts. Uses techniques like `COPY` statements, batch APIs, or staging files for maximum performance
upsert	Combined insert-or-update operation that handles conflicts when loading data	Essential for idempotent loading. Syntax varies by database: PostgreSQL `ON CONFLICT`, MySQL `ON DUPLICATE KEY`, SQL Server `MERGE`
schema mapping	Translation rules between source and destination data structures and types	Handles differences in column names, data types, and structural organization. May include type conversion rules and default value assignments
connection pooling	Reusing database connections across multiple operations for improved performance	Reduces connection overhead and manages concurrent access. Pools maintain minimum/maximum connection counts with timeout handling
incremental loading	Extracting and processing only data that has changed since the last pipeline execution	Core technique for efficient ETL at scale. Relies on watermarking, timestamps, or change data capture to identify new/modified records
lookback window	Small time buffer added to watermark queries to handle clock skew and late-arriving data	Prevents data loss from timing issues. Typically 1-5 minutes depending on system characteristics and consistency requirements
staging table	Temporary storage area for atomic bulk loading operations	Enables transactional loading patterns. Data is loaded to staging first, then atomically moved to final destination, allowing rollback on failure

Data Transformation Terms

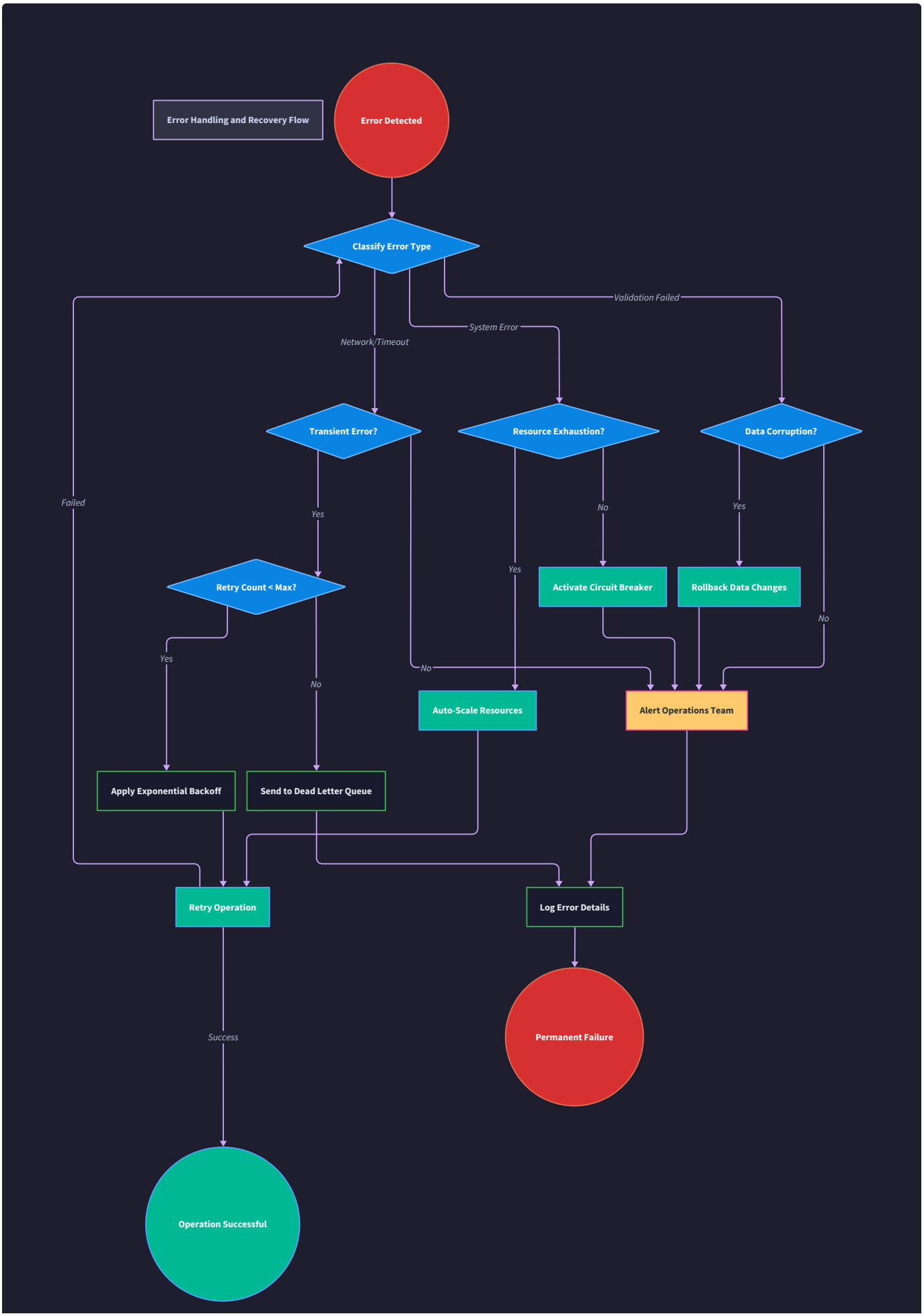
Term	Definition	Context and Usage
schema evolution	Systematic management of data structure changes over time while maintaining compatibility	Critical for production systems. Includes adding columns, changing types, and handling backward compatibility with existing consumers
type coercion	Automatic conversion between compatible data types during transformation	Can cause precision loss (float to int) or truncation. Requires careful handling with validation and error reporting for failed conversions
UDF	User-Defined Function - custom transformation logic written in Python or other languages	Enables complex business logic beyond SQL capabilities. Runs in isolated processes with proper error handling and resource limits
template rendering	Process of applying runtime parameters to SQL or configuration templates	Uses template engines like Jinja2 to generate final SQL queries. Enables parameterized pipelines with dynamic behavior based on runtime context
subprocess isolation	Running transformation code in separate processes for safety and resource control	Prevents memory leaks and crashes in one transformation from affecting others. Enables resource limits and timeout enforcement
schema registry	Centralized catalog of data structure definitions and their version history	Enables schema validation, evolution tracking, and compatibility checking. Supports multiple schema formats (JSON Schema, Avro, etc.)
validation pipeline	Series of data quality checks that records pass through during transformation	Includes type checking, constraint validation, and business rule verification. Failed records can be rejected, flagged, or sent to dead letter queues
null semantics	Rules for handling null/missing values across different systems and transformations	Varies significantly between databases and languages. Requires explicit handling in transformations to prevent unexpected behavior
compatibility checking	Verification that schema changes don't break existing pipeline consumers	Includes forward compatibility (new schemas work with old consumers) and backward compatibility (old schemas work with new consumers)

Pipeline Orchestration Terms

Term	Definition	Context and Usage
orchestration	Coordination of pipeline execution including scheduling, monitoring, and resource management	Encompasses the entire pipeline lifecycle from trigger through completion. Includes dependency resolution, parallel execution, and failure handling
scheduler	Component responsible for triggering pipeline execution based on time schedules or external events	Supports cron expressions for time-based triggers and event-driven execution. Manages schedule state and handles execution policies
execution engine	Component that runs pipeline tasks with parallelization, resource allocation, and state management	Core runtime system that executes tasks according to DAG dependencies. Manages task state transitions and resource allocation
state machine	Formal model defining valid task states and the events that trigger transitions between them	Ensures consistent task lifecycle management. States include PENDING, RUNNING, SUCCESS, FAILED with defined transition rules
resource allocation	Assignment of compute resources (CPU, memory, storage) to executing tasks	Prevents resource exhaustion and enables performance optimization. Includes resource estimation, reservation, and cleanup
metrics collection	Systematic gathering of performance and business metrics during pipeline execution	Enables monitoring, alerting, and performance optimization. Includes execution times, record counts, error rates, and resource usage
data lineage	Comprehensive tracking of data provenance and transformation history through pipelines	Records complete data flow from sources through transformations to destinations. Essential for compliance, debugging, and impact analysis
alert suppression	Preventing duplicate or cascading alerts during system outages or widespread issues	Reduces alert noise and prevents overwhelming operations teams. Uses correlation rules and time windows to group related alerts
level-based parallelization	Executing all tasks at the same DAG level simultaneously while respecting dependencies	Maximizes pipeline throughput by running independent tasks in parallel. Each level waits for the previous level to complete
message broker	Asynchronous communication system enabling loose coupling between pipeline components	Enables event-driven architecture and fault-tolerant communication. Common implementations include Apache Kafka, RabbitMQ, or cloud messaging services
dependency resolution	Process of determining task execution order based on declared prerequisites and constraints	Combines topological sorting with runtime conditions. Handles complex scenarios like conditional dependencies and dynamic task generation

Error Handling and Recovery Terms

Term	Definition	Context and Usage
exponential backoff	Retry strategy with exponentially increasing delays between attempts, often with randomization	Prevents overwhelming failing systems while providing reasonable retry behavior. Typical pattern: 1s, 2s, 4s, 8s with jitter
circuit breaker	Protection mechanism that stops calling a failing service to prevent cascading failures	Opens circuit after consecutive failures, enters half-open state for testing, closes when service recovers. Prevents thundering herd problems
dead letter queue	Storage system for messages that cannot be processed after maximum retry attempts	Enables manual inspection and reprocessing of failed items. Prevents data loss while avoiding infinite retry loops
jitter	Random variation added to retry timing to prevent synchronized load spikes	Prevents thundering herd when many clients retry simultaneously. Typically 10-50% random variation in delay timing
saga pattern	Breaking long-running transactions into smaller, compensatable steps with rollback capability	Enables fault tolerance in distributed systems. Each step has a corresponding compensation operation for rollback
two-phase commit	Distributed transaction protocol ensuring atomicity across multiple systems	Provides strong consistency guarantees at the cost of performance and availability. Requires all participants to vote before committing
compensation transaction	Reverse operation designed to undo the effects of a completed transaction step	Key component of saga pattern. Must be idempotent and handle partial completion scenarios


Error Handling and Recovery Flow
Error Detected
Classify Error Type
Validation Failed
System Error
Network/Timeout
Transient Error?
Resource Exhaustion?
Data Corruption?
Failed
Yes
No
Yes
Yes
No
Retry Count < Max?
Activate Circuit Breaker
Rollback Data Changes
Yes
No
No
No
Auto-Scale Resources
Alert Operations Team
Apply Exponential Backoff
Send to Dead Letter Queue
Retry Operation
Log Error Details
Success
Permanent Failure
Operation Successful

Monitoring and Operations Terms

Term	Definition	Context and Usage
health check	Automated verification of component operational status and readiness to handle requests	Includes connectivity tests, resource availability checks, and functional validation. Used by load balancers and monitoring systems
log correlation	Process of connecting related log entries across different components and time periods	Essential for distributed system debugging. Uses correlation IDs, timestamps, and context propagation to trace request flows
performance profiling	Systematic analysis of resource usage, bottlenecks, and optimization opportunities	Identifies CPU, memory, and I/O hotspots. Includes both real-time monitoring and historical analysis for capacity planning
symptom-cause mapping	Structured diagnostic approach that maps observable symptoms to underlying root causes	Systematic troubleshooting methodology. Documents known failure patterns and their resolution steps for faster incident response
interactive debugging	Real-time investigation of system behavior using debugging tools and techniques	Includes breakpoints, variable inspection, and step-through execution. Challenging in distributed systems due to timing dependencies
resource monitoring	Continuous tracking of system resource utilization including CPU, memory, disk, and network	Enables capacity planning, performance optimization, and early warning of resource exhaustion. Includes both host-level and application-level metrics
error context	Log entries and system state surrounding an error event to provide diagnostic information	Critical for effective troubleshooting. Includes events leading up to the error, concurrent activities, and system state at failure time

Advanced Architecture Terms

Term	Definition	Context and Usage
distributed execution	Running pipeline tasks across multiple machines or compute environments	Enables horizontal scaling and resource optimization. Requires coordination, state management, and fault tolerance across network boundaries
horizontal scaling	Adding more compute instances to handle increased load, as opposed to upgrading existing hardware	Preferred scaling approach for cloud systems. Requires stateless task design and effective load distribution mechanisms
auto-scaling	Automatically adjusting resource allocation based on current demand and performance metrics	Balances cost optimization with performance requirements. Includes scale-up triggers, scale-down policies, and resource estimation
cloud-native	Architecture designed specifically for cloud deployment patterns and services	Emphasizes containerization, microservices, and managed cloud services. Designed for elasticity, fault tolerance, and operational simplicity
stream processing	Continuous processing of data streams as they arrive, rather than batch processing	Enables real-time analytics and low-latency data pipelines. Requires different architectural patterns than traditional batch ETL
machine learning pipeline	Specialized workflow for ML model training, validation, and deployment with unique requirements	Includes data preprocessing, feature engineering, model training, validation, and deployment stages. Requires versioning and experiment tracking
real-time processing	Data processing with sub-second response time requirements	More demanding than stream processing, requiring specialized architectures and technologies. Often uses in-memory processing and optimized data structures
event-driven orchestration	Triggering pipeline execution based on external events rather than time schedules	Enables reactive data processing and just-in-time pipeline execution. Requires reliable event delivery and proper ordering
container orchestration	Managing containerized applications across clusters with scheduling, scaling, and service discovery	Enables portable deployment and efficient resource utilization. Common platforms include Kubernetes, Docker Swarm, and cloud container services
serverless integration	Using cloud functions and managed services for lightweight, event-driven tasks	Eliminates infrastructure management while providing automatic scaling. Suitable for simple transformations and integration tasks
checkpoint recovery	Resuming pipeline execution from saved processing state after failures	Enables fault tolerance without complete restart. Requires careful state management and atomic checkpoint operations
leader election	Choosing a primary instance in a distributed system to coordinate shared operations	Prevents split-brain scenarios and ensures single point of control. Common in distributed schedulers and coordination services
priority scheduling	Executing higher-priority tasks before lower-priority ones, subject to resource availability	Enables SLA management and critical path optimization. Requires careful balance to prevent starvation of lower-priority tasks

Data Types and Validation Terms

Term	Definition	Context and Usage
STRING	Variable-length text data type supporting Unicode characters	Most flexible data type but requires careful handling of encoding, length limits, and special characters during transformations
INTEGER	Whole number data type with defined precision and range limits	Common source of overflow errors during transformations. Different systems have different integer sizes and signedness
FLOAT	Floating-point numeric data type with inherent precision limitations	Can cause precision loss during type conversion. Requires careful handling of NaN, infinity, and rounding behaviors
DECIMAL	Fixed-precision numeric data type for exact decimal calculations	Preferred for financial calculations. Precision and scale must be preserved during transformations to prevent data loss
BOOLEAN	Binary true/false data type with varying representations across systems	Representations vary: true/false, 1/0, Y/N, T/F. Requires normalization during cross-system data movement
DATE	Calendar date without time component, with varying precision and timezone handling	Timezone-naive type that can cause issues in global systems. Requires careful handling of date arithmetic and comparisons
TIMESTAMP	Date and time data type with optional timezone information	Complex type requiring timezone handling, precision management, and careful comparison logic across systems
JSON	Semi-structured data type for nested objects and arrays	Requires schema validation and careful handling of type coercion for nested fields. Not all systems support JSON natively
BINARY	Raw binary data type for storing files, images, or encoded content	Requires base64 encoding for text-based transport. Large binary fields can impact pipeline performance

Task and Execution State Terms

Term	Definition	Context and Usage
PENDING	Initial task state before dependencies are evaluated and resources allocated	Task is defined but not yet ready for execution. Waiting for dependency resolution and resource availability
WAITING	Task state when dependencies are not yet satisfied	Task is ready to execute but blocked by upstream dependencies. Moves to QUEUED when dependencies complete successfully
QUEUED	Task state when ready for execution but waiting for available resources	Dependencies are satisfied but execution slot or resources not yet available. Managed by scheduler priority queues
RUNNING	Task state during active execution	Task is consuming resources and performing work. Requires monitoring for progress, timeouts, and resource usage
SUCCESS	Task completed successfully with expected outputs	Terminal state indicating successful completion. Enables downstream dependencies to begin execution
FAILED	Task completed unsuccessfully due to errors or exceptions	Terminal state unless retry policy applies. May trigger failure handling, alerts, and pipeline-level error responses
RETRYING	Task is scheduled for retry after a failure	Temporary state between failure and retry attempt. Managed by retry policy with exponential backoff timing
CANCELLED	Task execution was cancelled by user or system action	Terminal state for tasks that were stopped before completion. Requires cleanup of partial work and resources
SKIPPED	Task was intentionally bypassed due to conditional logic or upstream failures	Terminal state for tasks that didn't need to execute. Common in conditional pipelines and failure scenarios

Implementation Guidance

This glossary serves as both a learning resource and a practical reference during development. The terms are carefully chosen to align with industry standards while providing specific context for this ETL system implementation.

Technology Recommendations for Glossary Management

Component	Simple Option	Advanced Option
Glossary Storage	Static markdown files in repository	Searchable documentation platform (GitBook, Notion)
Term Cross-References	Manual links between sections	Automated link detection and validation
Definition Validation	Manual review process	Automated consistency checking against codebase
Usage Examples	Inline code snippets	Live examples from test cases

Glossary Maintenance Structure

```
project-root/
  docs/
    design/
      glossary.md                    ← this document
      term-usage-examples/           ← code examples for key terms
        dag-operations.py            ← DAG manipulation examples
        watermarking-patterns.py     ← incremental loading examples
        retry-policies.py            ← error handling examples
    api/
      terminology.json               ← machine-readable term definitions
      cross-references.json          ← term relationship mappings
  src/
    common/
      constants.py                   ← canonical constant definitions
      types.py                       ← type definitions matching glossary
```

Term Consistency Validation

The glossary terms should be validated against the actual codebase to ensure consistency. Key validation points include:

- Type names match exactly between glossary definitions and code declarations
- Method signatures align with interface descriptions
- Constants are defined with correct values and naming conventions
- Enum values match exactly across documentation and implementation
- State machine transitions are accurately reflected in both glossary and code

Common Glossary Usage Patterns

⚠️ **Pitfall: Terminology Drift** Terms defined in early design phases often evolve during implementation, leading to inconsistencies between documentation and code. Establish a single source of truth (preferably the code) and update documentation accordingly.

⚠️ **Pitfall: Overloaded Terms**
Some terms like "pipeline" or "state" have multiple meanings in different contexts. Always provide sufficient context to disambiguate, and consider using compound terms like "pipeline definition" vs "pipeline execution" when clarity is important.

⚠️ **Pitfall: Missing Domain Context** Technical terms often have different meanings in different domains. ETL-specific definitions may differ from general software engineering or database administration usage. Always provide ETL-specific context and usage examples.

Milestone Checkpoints

Checkpoint: Terminology Consistency

- Verify all type names in code match glossary definitions exactly
- Confirm method signatures align with interface descriptions
- Validate that state machine implementations match documented transitions
- Check that constant values match their glossary descriptions

Checkpoint: Documentation Completeness

- Ensure every public type has a corresponding glossary entry
- Verify all domain-specific terms are defined with appropriate context
- Confirm cross-references between related terms are accurate and helpful
- Validate that examples provided are current and functional

The glossary should be treated as a living document that evolves with the system while maintaining accuracy and usefulness as a reference tool for both learning and development activities.