

LLM Evaluation Framework: Design Document

Overview

A comprehensive system for evaluating Large Language Model applications through automated test case management, multi-metric scoring, and statistical analysis. The key architectural challenge is building a scalable, extensible evaluation pipeline that handles diverse metrics while maintaining result reproducibility and providing actionable insights for model performance optimization.

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): All milestones (1-4) - Understanding the evaluation challenge is foundational to all components

The LLM Evaluation Challenge

Think of evaluating a large language model like trying to grade a creative writing exam where the test questions are open-ended essays, there are multiple valid answers to each question, and the grading criteria themselves are subjective and context-dependent. Unlike traditional software testing where you can assert that `add(2, 3) == 5`, evaluating language models requires measuring qualities like coherence, factual accuracy, helpfulness, and style—none of which have simple binary answers.

The fundamental challenge stems from the **generative and probabilistic nature** of language models. Traditional software systems are deterministic: given the same input, they produce the same output every time. This makes testing straightforward—you define expected outputs and compare actual results. Language models, however, are designed to be creative and varied in their responses. The same prompt can yield dozens of different but equally valid answers, each with subtle differences in tone, structure, and approach.

Statistical evaluation becomes necessary because individual test cases don't provide reliable signals about model quality. A model might fail spectacularly on one carefully crafted prompt while excelling on a hundred similar ones. The evaluation system must aggregate results across large datasets to identify genuine patterns versus random variation. This requires sophisticated statistical analysis, confidence intervals, and significance testing—concepts foreign to traditional software QA processes.

The **multi-dimensional nature of language quality** adds another layer of complexity. A single response might be factually accurate but poorly written, or beautifully crafted but misleading. Traditional metrics like exact string matching are inadequate. Evaluation frameworks must simultaneously measure multiple aspects: factual correctness, grammatical quality, coherence, relevance, helpfulness, safety, and alignment with instructions. Each dimension may require different measurement approaches and may conflict with others.

Context dependency makes evaluation even more challenging. The quality of a language model response depends heavily on the specific use case, domain, and user expectations. A response that's excellent for a creative writing assistant

might be terrible for a medical information system. The evaluation framework must support domain-specific metrics, configurable quality criteria, and the ability to slice results by different contexts and use cases.

Scale and efficiency concerns emerge quickly in production environments. Comprehensive evaluation might involve thousands of test cases across dozens of metrics, each requiring expensive API calls to language models. The evaluation process itself often uses LLMs as judges, multiplying the computational cost. The framework must handle batch processing, parallel execution, caching, and graceful degradation when API limits are reached.

Critical Insight: Unlike traditional software where bugs are binary (the code works or it doesn't), language model evaluation deals with degrees of quality across multiple subjective dimensions. This requires statistical thinking, not deterministic assertions.

The **reproducibility challenge** is particularly acute for LLM evaluation. Language models are often non-deterministic by design, using temperature settings and sampling to introduce controlled randomness. External API-based models may change their behavior over time as providers update their systems. Evaluation results must be reproducible despite these sources of variation, requiring careful control of randomness, model versioning, and baseline management.

Human alignment presents the ultimate evaluation challenge. The goal isn't just technical correctness but alignment with human judgment and preferences. This requires expensive human annotation to establish ground truth, sophisticated techniques like LLM-as-judge evaluation where models evaluate other models, and continuous calibration against human preferences. The framework must support both automated metrics and human-in-the-loop evaluation workflows.

Current Evaluation Landscape

The current landscape of LLM evaluation tools reveals a fragmented ecosystem with significant gaps that prevent comprehensive, production-ready evaluation at scale. Understanding these limitations is crucial for designing an evaluation framework that addresses real-world needs rather than repeating existing mistakes.

Academic benchmarks and leaderboards dominate the current evaluation landscape, but they suffer from fundamental limitations when applied to production systems. Benchmarks like GLUE, SuperGLUE, and BIG-bench focus on narrow task-specific performance metrics that don't translate well to real-world applications. These benchmarks typically measure performance on standardized datasets with pre-defined metrics, but production LLM applications often require custom evaluation criteria that reflect specific business requirements and user needs.

The **static nature** of academic benchmarks creates additional problems. Models can be specifically optimized for benchmark performance without improving general capability, leading to Goodhart's law effects where "when a measure becomes a target, it ceases to be a good measure." Furthermore, benchmark datasets become contaminated over time as they're included in model training data, making historical comparisons unreliable. Production evaluation requires dynamic test suites that can evolve with the application and resist gaming.

Existing evaluation platforms like LangSmith, Weights & Biases, and various open-source tools provide pieces of the evaluation puzzle but lack comprehensive solutions. These tools typically excel in specific areas while leaving significant gaps in others.

Platform Category	Strengths	Limitations	Production Readiness
Academic Benchmarks	Standardized, reproducible, broad research adoption	Static datasets, narrow metrics, gaming susceptibility	Low - research focus
LangSmith/LangChain	Good dataset management, LLM integration, versioning	Limited metric types, vendor lock-in, scaling issues	Medium - emerging platform
W&B/ML Ops Platforms	Excellent experiment tracking, visualization, collaboration	Generic ML focus, limited LLM-specific features	Medium - requires customization
Open Source Tools	Customizable, transparent, community-driven	Fragmented, maintenance burden, limited support	Low - requires significant investment
Custom Solutions	Perfect fit for specific needs, full control	High development cost, maintenance burden, expertise required	High - but expensive

Metric limitations represent another critical gap in the current landscape. Most existing tools rely heavily on traditional NLP metrics like BLEU and ROUGE, which measure surface-level text similarity but poorly capture semantic meaning, factual accuracy, or task-specific quality. While newer approaches like BERTScore and semantic similarity using embeddings show promise, they lack calibration and interpretability. Practitioners struggle to understand what constitutes a "good" semantic similarity score for their specific use case.

The **LLM-as-judge** approach has emerged as a popular solution for more nuanced evaluation, where language models evaluate the outputs of other language models. However, current implementations suffer from inconsistency, bias, and lack of proper calibration. Different judge models can produce dramatically different scores for the same content, and there's insufficient tooling for ensuring judge reliability, detecting judge bias, or calibrating judge scores against human preferences.

Scalability and cost management represent practical barriers that existing tools inadequately address. Comprehensive evaluation can require thousands of API calls, quickly becoming expensive and slow. Most tools lack sophisticated caching, batching, and rate limiting capabilities. They also struggle with the checkpoint and resume functionality necessary for large-scale evaluations that may be interrupted by API failures or resource constraints.

Integration and workflow challenges prevent many organizations from adopting systematic evaluation practices. Existing tools often require significant integration work to fit into production workflows. They lack the CI/CD integration, automated regression detection, and reporting capabilities necessary for embedding evaluation into software development processes. Many tools assume data scientists will manually run evaluations rather than supporting automated, continuous evaluation pipelines.

Decision: Build Comprehensive Framework Rather Than Extend Existing Tools

- **Context:** Organizations need production-ready evaluation capabilities that current tools don't provide comprehensively
- **Options Considered:**
 1. Extend existing open-source tools with missing capabilities
 2. Integrate multiple existing tools into a unified workflow
 3. Build new comprehensive framework from scratch
- **Decision:** Build new framework with clean architecture and comprehensive feature set
- **Rationale:** Existing tools have fundamental architectural limitations that prevent adding missing capabilities without major rewrites. Integration approaches create maintenance nightmares and still leave capability gaps. A purpose-built framework can address all requirements with consistent architecture.
- **Consequences:** Higher initial development cost but better long-term maintainability, feature completeness, and user experience. Avoids vendor lock-in and technical debt from working around existing tool limitations.

Data management and versioning capabilities in existing tools are often inadequate for production needs. While tools like LangSmith provide basic dataset versioning, they lack the sophisticated diff, merge, and rollback capabilities necessary for managing evolving test suites in collaborative environments. They also struggle with large datasets, lacking efficient storage, querying, and sampling capabilities.

Statistical rigor is notably absent from most existing evaluation tools. They provide basic aggregation (mean, median) but lack the statistical testing, confidence intervals, and significance analysis necessary for making reliable comparisons between models or detecting genuine performance regressions versus random variation. This limitation prevents organizations from making data-driven decisions about model deployments and improvements.

The **reporting and analysis** capabilities of current tools focus primarily on displaying individual scores rather than providing actionable insights. They lack sophisticated failure analysis, error clustering, and root cause analysis capabilities. Reports are often generic rather than tailored to specific stakeholders' needs (engineers vs. product managers vs. executives).

Customization and extensibility represent final major limitations. While most tools claim to support custom metrics, the reality is often clunky plugin systems, limited documentation, and poor integration with the broader evaluation workflow. Organizations end up building significant custom tooling anyway, defeating the purpose of using existing platforms.

⚠ Pitfall: Evaluation Tool Selection Based on Demos Rather Than Production Needs Many organizations select evaluation tools based on impressive demos or marketing materials without thoroughly evaluating production requirements. They discover limitations only after significant integration work, leading to tool abandonment or expensive workarounds. Always evaluate tools against comprehensive production checklists including scale, customization, integration, and statistical capabilities.

The market clearly needs a comprehensive evaluation framework that combines the dataset management capabilities of platforms like LangSmith, the statistical rigor of academic research tools, the scalability requirements of production systems, and the customization capabilities that organizations require for their specific use cases. This framework must be architecturally designed from the ground up to handle the unique challenges of LLM evaluation rather than adapting tools designed for traditional machine learning workflows.

The **emergence of foundation model evaluation** as a distinct discipline requires specialized tooling that understands the unique characteristics of language models: their probabilistic outputs, multi-dimensional quality criteria, context dependency, and the need for both automated and human evaluation. Current tools approach LLM evaluation as an extension of traditional ML evaluation, missing opportunities for LLM-specific optimizations and capabilities.

Implementation Guidance

The complexity of LLM evaluation requires careful technology selection and implementation planning. This guidance provides concrete recommendations for building a production-ready evaluation framework that addresses the challenges identified above.

A. Technology Recommendations:

Component	Simple Option	Advanced Option	Recommended for Production
Dataset Storage	JSON files + Git	PostgreSQL + Alembic migrations	PostgreSQL for scale, JSON for prototypes
Metrics Computation	Scikit-learn + NLTK	HuggingFace Transformers + custom CUDA kernels	HuggingFace ecosystem for flexibility
LLM API Integration	OpenAI client + requests	LiteLLM unified interface + async clients	LiteLLM for multi-provider support
Caching Layer	Pickle files + file locks	Redis with persistence	Redis for production, files for development
Statistical Analysis	SciPy + NumPy	Statsmodels + Pandas + Plotly	Statsmodels for rigorous analysis
Report Generation	Jinja2 templates + Matplotlib	React dashboard + D3.js visualizations	Jinja2 for simplicity, React for interactivity
Task Orchestration	Threading + asyncio	Celery + Redis	Asyncio for I/O bound, Celery for complex workflows

B. Recommended Project Structure:

```
llm_evaluation_framework/
├── setup.py                                # Package configuration and dependencies
├── requirements.txt                          # Python dependencies
├── pyproject.toml                           # Modern Python project configuration
├── README.md                                # Quick start guide and examples
├── docs/
│   ├── architecture.md                      # Comprehensive documentation
│   ├── metrics_guide.md                     # System architecture overview
│   └── deployment.md                        # Guide to built-in and custom metrics
│   └── deployment.md                        # Production deployment guide
├── src/llm_eval/
│   ├── __init__.py                           # Main package source
│   ├── core/
│   │   ├── __init__.py                      # Package initialization and version
│   │   ├── test_case.py                     # Core abstractions and interfaces
│   │   ├── metric.py                        # TestCase and Dataset model definitions
│   │   ├── evaluation.py                   # Base metric interface and registry
│   │   └── exceptions.py                  # EvaluationRun and result models
│   │   └── exceptions.py                  # Custom exception hierarchy
│   ├── dataset/
│   │   ├── __init__.py                      # Dataset management (Milestone 1)
│   │   ├── loader.py                        # Multi-format dataset loading
│   │   ├── versioning.py                   # Git-like versioning system
│   │   ├── splitter.py                      # Train/test/validation splitting
│   │   └── validators.py                  # Schema validation and data quality checks
│   ├── metrics/
│   │   ├── __init__.py                      # Metrics engine (Milestone 2)
│   │   ├── exact_match.py                  # Exact and fuzzy string matching
│   │   ├── semantic.py                     # Embedding-based similarity metrics
│   │   ├── llm_judge.py                     # LLM-as-judge implementation
│   │   ├── traditional.py                 # BLEU, ROUGE, BERTScore
│   │   └── custom.py                        # Plugin system for custom metrics
│   ├── runner/
│   │   ├── __init__.py                      # Evaluation execution (Milestone 3)
│   │   ├── executor.py                     # Main evaluation orchestration
│   │   ├── llm_client.py                   # Unified LLM API client wrapper
│   │   ├── cache.py                        # Response caching system
│   │   ├── checkpoint.py                  # Progress checkpointing and recovery
│   │   └── progress.py                     # Progress tracking and reporting
│   ├── analysis/
│   │   ├── __init__.py                      # Reporting and analysis (Milestone 4)
│   │   ├── aggregation.py                 # Statistical aggregation and summaries
│   │   ├── regression.py                  # Automated regression detection
│   │   ├── clustering.py                   # Failure pattern clustering
│   │   ├── visualization.py               # Chart and graph generation
│   │   └── reporting.py                   # HTML/PDF report generation
│   └── utils/
│       ├── __init__.py                      # Shared utilities
│       ├── logging.py                      # Structured logging configuration
│       ├── config.py                       # Configuration management
│       └── async_utils.py                 # Async/await helper functions
└── tests/
    ├── unit/
    ├── integration/
    ├── end_to_end/
    ├── fixtures/
    └── conftest.py                         # Comprehensive test suite
    └── conftest.py                         # Unit tests for individual components
    └── fixtures/                           # Integration tests between components
    └── fixtures/                           # Full workflow tests
    └── fixtures/                           # Test data and mock responses
    └── fixtures/                           # Pytest configuration and fixtures
└── examples/
    ├── quick_start.py                     # Example usage and tutorials
    ├── custom_metrics.py                  # 5-minute getting started example
    └── production_workflow.py            # How to implement custom metrics
    └── production_workflow.py            # Complete production evaluation workflow
```

```
|   └── datasets/          # Sample evaluation datasets
└── scripts/              # Development and deployment scripts
    ├── setup_dev.py       # Development environment setup
    ├── run_benchmarks.py  # Performance benchmarking
    └── deploy.py          # Production deployment automation
```

C. Core Infrastructure Starter Code:

The following provides complete, production-ready infrastructure components that handle cross-cutting concerns, allowing developers to focus on the core evaluation logic.

Configuration Management (`src/llm_eval/utils/config.py`):

```
"""Configuration management with environment variables and validation."""

import os

from dataclasses import dataclass, field

from typing import Dict, List, Optional, Union

from pathlib import Path

import yaml

import json

@dataclass

class LLMProviderConfig:

    """Configuration for LLM API providers."""

    provider: str # "openai", "anthropic", "local"

    api_key: Optional[str] = None

    base_url: Optional[str] = None

    model: str = "gpt-3.5-turbo"

    max_tokens: int = 1024

    temperature: float = 0.0

    timeout: int = 30

    max_retries: int = 3

    rate_limit_per_minute: int = 60

@dataclass

class EvaluationConfig:

    """Main configuration for evaluation framework."""

    # Dataset configuration

    dataset_storage_path: Path = field(default_factory=lambda: Path("./datasets"))

    max_dataset_size_mb: int = 1000

    # Execution configuration

    max_concurrent_evaluations: int = 10

    batch_size: int = 32
```

```
cache_ttl_hours: int = 24

enable_checkpointing: bool = True

checkpoint_interval: int = 50

# LLM provider configurations

llm_providers: Dict[str, LLMPProviderConfig] = field(default_factory=dict)

default_provider: str = "openai"

# Analysis configuration

confidence_level: float = 0.95

min_sample_size_for_significance: int = 30

regression_threshold: float = 0.05

# Reporting configuration

output_format: str = "html" # "html", "pdf", "json"

include_visualizations: bool = True

max_report_size_mb: int = 50

@classmethod

def from_file(cls, config_path: Union[str, Path]) -> 'EvaluationConfig':

    """Load configuration from YAML or JSON file."""

    config_path = Path(config_path)

    if not config_path.exists():

        raise FileNotFoundError(f"Configuration file not found: {config_path}")

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

        if config_path.suffix.lower() == '.yaml' or config_path.suffix.lower() == '.yml':

            data = yaml.safe_load(f)

        elif config_path.suffix.lower() == '.json':

            data = json.load(f)
```

```

    else:

        raise ValueError(f"Unsupported config file format: {config_path.suffix}")

    # Convert nested provider configs

    if 'llm_providers' in data:

        providers = {}

        for name, config in data['llm_providers'].items():

            providers[name] = LLMPProviderConfig(**config)

        data['llm_providers'] = providers

    return cls(**data)

@classmethod

def from_environment(cls) -> 'EvaluationConfig':

    """Create configuration from environment variables."""

    # TODO: Implement environment variable parsing

    # Look for EVAL_* environment variables and map to config fields

    # Example: EVAL_MAX_CONCURRENT=20 -> max_concurrent_evaluations=20

    pass

def get_default_config() -> EvaluationConfig:

    """Get default configuration with common LLM providers pre-configured."""

    return EvaluationConfig(
        llm_providers={
            "openai": LLMPProviderConfig(
                provider="openai",
                api_key=os.getenv("OPENAI_API_KEY"),
                model="gpt-3.5-turbo"
            ),
            "anthropic": LLMPProviderConfig(

```

```
        provider="anthropic",  
        api_key=os.getenv("ANTHROPIC_API_KEY"),  
        model="claude-3-sonnet-20240229"  
    )  
,  
    default_provider="openai"  
)
```

Async Utilities (`src/llm_eval/utils/async_utils.py`):

```
"""Async utilities for handling concurrent LLM API calls with rate limiting."""

import asyncio

import time

from typing import List, Callable, TypeVar, Awaitable, Any, Optional

from dataclasses import dataclass

import logging

T = TypeVar('T')

logger = logging.getLogger(__name__)

@dataclass

class RateLimiter:

    """Token bucket rate limiter for API calls."""

    max_requests: int

    time_window: int = 60  # seconds

    _tokens: int = field(init=False)

    _last_refill: float = field(init=False)

    def __post_init__(self):

        self._tokens = self.max_requests

        self._last_refill = time.time()

    async def acquire(self) -> None:

        """Acquire a token, waiting if necessary."""

        now = time.time()

        elapsed = now - self._last_refill

        # Refill tokens based on elapsed time

        tokens_to_add = int(elapsed * self.max_requests / self.time_window)

        self._tokens = min(self.max_requests, self._tokens + tokens_to_add)

        self._last_refill = now
```

```
if self._tokens > 0:

    self._tokens -= 1

    return


# Wait until we can get a token

wait_time = self.time_window / self.max_requests

await asyncio.sleep(wait_time)

await self.acquire()


class BatchProcessor:

    """Process items in batches with concurrency control and rate limiting."""

    def __init__(

        self,
        batch_size: int = 32,
        max_concurrent: int = 10,
        rate_limiter: Optional[RateLimiter] = None
    ):

        self.batch_size = batch_size

        self.semaphore = asyncio.Semaphore(max_concurrent)

        self.rate_limiter = rate_limiter


    async def process_batch(

        self,
        items: List[T],
        processor_func: Callable[[T], Awaitable[Any]],
        progress_callback: Optional[Callable[[int, int], None]] = None
    ) -> List[Any]:

        """Process items in parallel batches."""

```

```
results = []

total_items = len(items)

for i in range(0, total_items, self.batch_size):

    batch = items[i:i + self.batch_size]

    batch_results = await self._process_single_batch(batch, processor_func)

    results.extend(batch_results)

    if progress_callback:

        completed = min(i + self.batch_size, total_items)

        progress_callback(completed, total_items)

return results

async def _process_single_batch(

    self,

    batch: List[T],

    processor_func: Callable[[T], Awaitable[Any]]

) -> List[Any]:

    """Process a single batch with semaphore and rate limiting."""

    async def process_with_limits(item: T) -> Any:

        async with self.semaphore:

            if self.rate_limiter:

                await self.rate_limiter.acquire()

            return await processor_func(item)

    tasks = [process_with_limits(item) for item in batch]

    return await asyncio.gather(*tasks, return_exceptions=True)

# TODO: Add retry logic with exponential backoff
```

```
# TODO: Add circuit breaker pattern for failing APIs  
  
# TODO: Add request/response logging for debugging  
  
# TODO: Add metrics collection (latency, success rate, etc.)
```

D. Core Logic Skeletons:

The following skeletons provide the structure for core components that developers should implement, with detailed TODOs mapping to the design concepts.

TestCase and Dataset Models (`src/llm_eval/core/test_case.py`):

```
"""Core data models for test cases and datasets."""

from dataclasses import dataclass, field

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

from datetime import datetime

from pathlib import Path

import uuid

@dataclass

class TestCase:

    """Individual test case for LLM evaluation."""

    def __init__(

        self,
        prompt: str,
        expected_output: str,
        tags: List[str] = None,
        metadata: Dict[str, Any] = None,
        difficulty: str = "medium",
        case_id: str = None
    ):

        # TODO 1: Validate prompt is non-empty string

        # TODO 2: Validate expected_output is non-empty string

        # TODO 3: Validate tags list contains only string values

        # TODO 4: Validate difficulty is one of: "easy", "medium", "hard"

        # TODO 5: Generate UUID for case_id if not provided

        # TODO 6: Initialize metadata as empty dict if None

        # TODO 7: Store all validated values as instance attributes

        pass

    def to_dict(self) -> Dict[str, Any]:
```

```
"""Convert test case to dictionary for serialization."""

# TODO 1: Create dictionary with all instance attributes

# TODO 2: Ensure datetime objects are converted to ISO strings

# TODO 3: Return dictionary suitable for JSON serialization

pass


@classmethod

def from_dict(cls, data: Dict[str, Any]) -> 'TestCase':

    """Create test case from dictionary."""

    # TODO 1: Extract required fields (prompt, expected_output)

    # TODO 2: Extract optional fields with defaults

    # TODO 3: Validate all fields using same logic as __init__

    # TODO 4: Return new TestCase instance

    pass


@dataclass

class Dataset:

    """Collection of test cases with versioning and metadata."""

    def __init__(

        self,
        name: str,
        test_cases: List[TestCase] = None,
        version: str = "1.0.0",
        description: str = "",
        created_at: datetime = None
    ):

        # TODO 1: Validate name is non-empty and follows naming convention

        # TODO 2: Initialize test_cases as empty list if None

        # TODO 3: Validate version follows semantic versioning format
```

```
# TODO 4: Set created_at to current datetime if None

# TODO 5: Store all values as instance attributes

# TODO 6: Initialize empty _index for fast case lookup by ID

pass

def add_test_case(self, test_case: TestCase) -> None:

    """Add a test case to the dataset."""

    # TODO 1: Validate test_case is TestCase instance

    # TODO 2: Check for duplicate case_id in existing test cases

    # TODO 3: Add test_case to internal list

    # TODO 4: Update lookup index for fast retrieval

    pass

def get_test_case(self, case_id: str) -> Optional[TestCase]:

    """Retrieve test case by ID."""

    # TODO 1: Look up case_id in the index

    # TODO 2: Return TestCase if found, None otherwise

    pass

def filter_by_tags(self, tags: List[str]) -> 'Dataset':

    """Create new dataset with test cases matching any of the given tags."""

    # TODO 1: Find test cases where any tag in test_case.tags matches tags list

    # TODO 2: Create new Dataset with filtered test cases

    # TODO 3: Preserve metadata but update name to indicate filtering

    # TODO 4: Return new Dataset instance

    pass

def train_test_split(
    self,
```

```
    test_size: float = 0.2,
    stratify_by: Optional[str] = None,
    random_seed: int = 42
) -> tuple['Dataset', 'Dataset']:
    """Split dataset into train and test sets."""

    # TODO 1: Validate test_size is between 0 and 1

    # TODO 2: If stratify_by provided, group test cases by that metadata field

    # TODO 3: Use random_seed for reproducible splitting

    # TODO 4: Ensure both train and test sets have representative samples

    # TODO 5: Return tuple of (train_dataset, test_dataset)

    pass

# TODO: Add DatasetVersion class for tracking changes over time

# TODO: Add DatasetSplit class for managing train/test/validation divisions

# TODO: Add validation methods for schema compliance

# TODO: Add export methods for different formats (CSV, JSON, JSONL)
```

Base Metric Interface (`src/llm_eval/core/metric.py`):

```
"""Base metric interface and registry system."""

from abc import ABC, abstractmethod

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

from dataclasses import dataclass

import logging

@dataclass

class MetricResult:

    """Result of applying a metric to model output."""

    score: float # Normalized score between 0.0 and 1.0

    metadata: Dict[str, Any] = field(default_factory=dict)

    explanation: Optional[str] = None

    def __post_init__(self):

        """Validate metric result after creation."""

        # TODO 1: Ensure score is numeric (int or float)

        # TODO 2: Ensure score is between 0.0 and 1.0 inclusive

        # TODO 3: Raise ValueError with clear message if validation fails

        pass

class BaseMetric(ABC):

    """Abstract base class for all evaluation metrics."""

    def __init__(self, name: str, description: str = ""):

        # TODO 1: Validate name is non-empty string

        # TODO 2: Store name and description as instance attributes

        # TODO 3: Initialize any metric-specific configuration

        pass

    @abstractmethod

    async def compute(
```

```

        self,
        model_output: str,
        expected_output: str,
        test_case: 'TestCase'
    ) -> MetricResult:
        """Compute metric score for model output against expected output."""
        # TODO: Each metric subclass must implement this method
        # TODO: Return MetricResult with score between 0.0 and 1.0
        # TODO: Include explanation of how score was calculated
        # TODO: Handle edge cases (empty strings, None values, etc.)
        pass

    @abstractmethod
    def is_applicable(self, test_case: 'TestCase') -> bool:
        """Check if this metric is applicable to the given test case."""
        # TODO: Return True if metric can meaningfully evaluate this test case
        # TODO: Consider test case tags, difficulty, or other metadata
        # TODO: Example: LLM-judge might not apply to math problems
        pass

    def preprocess_output(self, text: str) -> str:
        """Preprocess model output before scoring (override if needed)."""
        # TODO 1: Strip leading and trailing whitespace
        # TODO 2: Normalize whitespace (multiple spaces -> single space)
        # TODO 3: Apply any metric-specific preprocessing
        # TODO 4: Return cleaned text
        return text.strip()

    def preprocess_expected(self, text: str) -> str:

```

```
"""Preprocess expected output before scoring (override if needed)."""

# TODO: Apply same preprocessing as model output for fair comparison

return self._preprocess_output(text)

class MetricRegistry:

"""Registry for managing available metrics."""

def __init__(self):

    self._metrics: Dict[str, BaseMetric] = {}

    self._metric_classes: Dict[str, type] = {}

def register_metric(self, metric: BaseMetric) -> None:

    """Register a metric instance."""

    # TODO 1: Validate metric is BaseMetric instance

    # TODO 2: Check for name conflicts with existing metrics

    # TODO 3: Store metric in internal registry

    # TODO 4: Log successful registration

    pass

def register_metric_class(self, name: str, metric_class: type) -> None:

    """Register a metric class for lazy instantiation."""

    # TODO 1: Validate metric_class is subclass of BaseMetric

    # TODO 2: Store class in registry for later instantiation

    # TODO 3: Allow factory-style metric creation

    pass

def get_metric(self, name: str) -> Optional[BaseMetric]:

    """Get metric by name."""

    # TODO 1: Look up metric in registry

    # TODO 2: If not found as instance, try creating from class
```

```

# TODO 3: Return metric instance or None if not found

pass


def list_metrics(self) -> List[str]:
    """List all registered metric names."""

    # TODO: Return sorted list of all available metric names

    pass


def get_applicable_metrics(self, test_case: 'TestCase') -> List[BaseMetric]:
    """Get all metrics applicable to a test case."""

    # TODO 1: Get all registered metrics

    # TODO 2: Filter to only those where is_applicable(test_case) is True

    # TODO 3: Return list of applicable metric instances

    pass


# Global metric registry instance

METRIC_REGISTRY = MetricRegistry()


# TODO: Add metric combination support (weighted averages, ensembles)

# TODO: Add metric caching to avoid recomputing expensive scores

# TODO: Add metric validation to ensure output is always 0.0-1.0 range

# TODO: Add async batch processing for metrics that can be batched

```

E. Language-Specific Hints:

Python Development Environment:

- Use `python -m venv venv` to create isolated environment for dependencies
- Install development dependencies: `pip install pytest pytest-asyncio black isort mypy`
- Use `black` for code formatting: `black --line-length 88 src/`
- Use `mypy` for type checking: `mypy src/ --strict`
- Use `pytest-asyncio` for testing async functions: `@pytest.mark.asyncio`

Async/Await Patterns:

- Always use `async def` for functions that make API calls or I/O operations
- Use `asyncio.gather()` for running multiple coroutines concurrently

- Use `asyncio.Semaphore(n)` to limit concurrent operations to `n`
- Use `aiohttp.ClientSession()` for HTTP requests instead of `requests`
- Handle `asyncio.TimeoutError` and `aiohttp.ClientError` exceptions

Error Handling Best Practices:

- Create custom exception hierarchy in `src/llm_eval/core/exceptions.py`
- Use specific exception types: `DatasetValidationError`, `MetricComputationError`, etc.
- Always include original error in exception chains: `raise NewError() from original_error`
- Log errors with structured data: `logger.error("Metric failed", extra={"test_case_id": case_id, "metric": metric_name})`

Performance Optimization:

- Use `functools.lru_cache()` for expensive pure functions
- Use `asyncio.create_task()` to start background tasks early
- Batch API requests when possible rather than making individual calls
- Use generators for large datasets: `yield` instead of building large lists
- Profile with `cProfile` and `py-spy` to identify bottlenecks

F. Milestone Checkpoints:

After Understanding Context (Current Milestone):

- **Verification Command:** Review existing evaluation tools in your domain
- **Expected Understanding:** Clear grasp of why LLM evaluation differs from traditional testing
- **Manual Check:** Can you explain to a colleague why BLEU score isn't sufficient for chatbot evaluation?
- **Red Flags:** If you think evaluation is "just comparing strings," revisit the statistical evaluation concepts

Before Starting Implementation:

- **Setup Verification:** `python -c "import asyncio, aiohttp, pandas, numpy; print('Dependencies OK')"`
- **Architecture Check:** Draw the four-layer architecture from memory (Dataset → Metrics → Runner → Reports)
- **Concept Check:** Can you list 5 different types of evaluation metrics and when to use each?

The foundation established in this context section is crucial for making informed decisions throughout the implementation. The challenges identified here will resurface in every component design, and the landscape analysis informs the architectural choices made in subsequent sections.

Goals and Non-Goals

Milestone(s): All milestones (1-4) - Goal definition drives design decisions across dataset management, metrics, evaluation execution, and reporting

Primary Goals

Think of this evaluation framework as a **comprehensive testing laboratory** for language models. Just as a medical laboratory provides standardized tests with reliable results that doctors can trust for diagnosis, our framework must

provide standardized evaluation protocols that AI engineers can trust for model assessment. The laboratory analogy extends further: we need proper sample handling (dataset management), various diagnostic tests (metrics), efficient processing (parallel evaluation), and clear reports that enable actionable decisions.

The framework's **primary responsibility** is to transform the chaotic, ad-hoc process of LLM evaluation into a systematic, reproducible, and scalable workflow. Currently, most teams evaluate their language models through manual spot-checking or simple accuracy calculations on small samples. This approach fails catastrophically as applications grow in complexity and scale. Our framework bridges this gap by providing enterprise-grade evaluation infrastructure that maintains scientific rigor while remaining accessible to practitioners.

Core Evaluation Capabilities

The framework must deliver **comprehensive evaluation coverage** across the full spectrum of language model applications. This means supporting both objective metrics like exact string matching and subjective metrics that require semantic understanding. The system treats evaluation as a multi-dimensional problem where different aspects of model performance require different measurement approaches.

Objective measurement capabilities form the foundation of the evaluation system. The framework implements exact match scoring with configurable normalization options, allowing teams to specify whether whitespace, capitalization, or punctuation differences should be considered errors. Fuzzy matching extends this capability by using edit distance algorithms to measure how close model outputs are to expected results, with configurable similarity thresholds.

Semantic evaluation capabilities address the fundamental challenge that language allows multiple correct expressions of the same meaning. The framework implements semantic similarity scoring using embedding models to compare the meaning of model outputs against reference answers. This approach captures cases where the model produces a correct answer using different words or phrasing than the expected output.

Statistical evaluation robustness ensures that evaluation results are scientifically sound and actionable. The framework computes confidence intervals for all metrics, performs significance testing when comparing model versions, and adjusts for multiple comparisons to prevent false discoveries. This statistical rigor transforms evaluation from guesswork into evidence-based decision making.

Design Insight: The key architectural decision here is treating metrics as first-class plugins rather than hardcoded functions. This enables teams to implement domain-specific evaluation criteria while leveraging the framework's infrastructure for execution, caching, and reporting.

Evaluation Capability	Implementation Approach	Supported Use Cases	Quality Assurance
Exact Match	Character-by-character comparison with normalization options	Structured output validation, code generation, factual QA	Configurable whitespace/case handling
Fuzzy Match	Edit distance with configurable similarity thresholds	Natural language answers with minor variations	Threshold calibration against human judgment
Semantic Similarity	Embedding cosine similarity with multiple embedding models	Open-ended responses, paraphrasing tasks	Cross-validation against human evaluators
LLM-as-Judge	Structured prompting of judge models with consistency controls	Complex reasoning, creativity, subjective quality	Multi-judge consensus and calibration
Custom Metrics	Plugin architecture with standardized interfaces	Domain-specific evaluation criteria	Validation against ground truth samples

Dataset Management and Versioning

Dataset management operates like a version control system specifically designed for evaluation data. Think of it as "Git for test cases" where every change to the evaluation dataset is tracked, reversible, and auditable. This versioning capability is crucial because evaluation datasets evolve continuously as teams discover edge cases, add new test scenarios, or refine expected outputs based on model behavior.

The framework implements **immutable dataset versioning** where each version represents a complete snapshot of the evaluation data at a specific point in time. This approach prevents the common problem of evaluation drift, where gradual changes to test cases make it impossible to compare model performance across time periods. Teams can confidently compare a model's performance today against its performance six months ago using the exact same evaluation criteria.

Dataset splitting capabilities ensure proper train/test separation with stratified sampling across important dimensions like difficulty level, topic category, or input length. The framework prevents data leakage by maintaining strict boundaries between training and evaluation data, with cryptographic hashing to detect any overlap between splits.

Dataset Management Feature	Technical Implementation	Prevents Common Problem	Audit Capability
Version Control	Content-addressable storage with merkle trees	Evaluation drift over time	Full change history with diffs
Dataset Splitting	Stratified sampling with configurable ratios	Biased evaluation samples	Reproducible splits with fixed seeds
Schema Validation	JSON Schema with custom validators	Malformed test cases	Validation error reports
Import/Export	Multi-format support with automatic mapping	Data format lock-in	Provenance tracking
Golden Examples	Quality-weighted sampling for calibration	Inconsistent evaluation standards	Human annotation tracking

Scalable Execution Infrastructure

The evaluation runner operates as a **distributed task execution engine** optimized for language model API calls. Think of it as a specialized job scheduler that understands the unique characteristics of LLM evaluation: high latency, rate limits, variable response times, and expensive API calls that should never be repeated unnecessarily.

Parallel execution capabilities maximize throughput while respecting API rate limits through intelligent backoff strategies and request queuing. The framework automatically distributes evaluation work across available resources, scaling from single-machine development environments to distributed cloud deployments.

Caching and checkpointing transform evaluation from a fragile, all-or-nothing process into a robust, resumable workflow. The framework caches all model responses using content-based hashing, ensuring that re-running evaluations with identical inputs skips expensive API calls. Checkpoint files enable recovery from failures without losing progress, critical for large-scale evaluations that may take hours or days to complete.

Design Insight: The caching strategy uses content-addressable storage where the cache key combines the model configuration, prompt, and generation parameters. This approach automatically invalidates cached results when any evaluation parameter changes while maximizing cache hits for identical requests.

Comprehensive Reporting and Analysis

The reporting system transforms raw evaluation scores into **actionable insights** that guide model improvement decisions. Rather than simply presenting aggregate scores, the framework performs statistical analysis to identify performance patterns, flag significant changes, and highlight areas needing attention.

Regression detection automatically compares current evaluation results against historical baselines, flagging metrics that have degraded beyond statistical significance thresholds. This capability enables continuous integration workflows where model changes that significantly harm performance trigger alerts or block deployments.

Failure analysis clusters common error patterns and identifies the test cases where models struggle most. This analysis helps teams understand whether model failures are concentrated in specific domains, input types, or difficulty levels, enabling targeted improvement efforts.

Decision: Comprehensive Goal Scope

- **Context:** LLM evaluation spans many different use cases, from simple classification to complex reasoning tasks, each requiring different measurement approaches
- **Options Considered:**
 1. Focus on a narrow set of metrics for simplicity
 2. Build a comprehensive framework supporting diverse evaluation needs
 3. Create a minimal core with extensive plugin architecture
- **Decision:** Comprehensive framework with plugin architecture for extensibility
- **Rationale:** Production AI systems require diverse evaluation approaches, and teams need a single platform that grows with their sophistication rather than multiple specialized tools
- **Consequences:** Higher initial complexity but significantly better long-term adoption and utility across different use cases

Explicit Non-Goals

Setting clear boundaries prevents **scope creep** that could compromise the framework's core mission. Think of these non-goals as **architectural guardrails** that maintain focus on evaluation excellence rather than expanding into adjacent but distinct problem domains.

Model Training and Fine-Tuning

The framework explicitly **does not implement model training capabilities**. While evaluation results inform training decisions, the framework treats model training as a separate concern handled by specialized ML training platforms. This separation maintains clean architectural boundaries and prevents the framework from becoming an unwieldy all-in-one ML platform.

Training data preparation falls outside the framework's scope, even though evaluation datasets might inform training data selection. Teams use external tools for data cleaning, augmentation, and training set preparation, then import the results into the evaluation framework for assessment.

Hyperparameter optimization represents another boundary. While evaluation results guide hyperparameter selection, the framework does not implement automated hyperparameter search algorithms. Teams integrate the framework with external optimization tools when needed.

Design Insight: This boundary prevents the common antipattern of evaluation frameworks becoming bloated ML platforms that do many things poorly instead of doing evaluation exceptionally well.

Real-Time Model Serving

The framework **does not provide model serving infrastructure** for production deployments. Evaluation runs in batch mode against pre-trained models accessed through APIs or local inference, but the framework does not implement real-time serving, load balancing, or production monitoring capabilities.

Production monitoring represents a related but separate concern. While the framework can evaluate production model outputs in batch mode, it does not provide real-time monitoring, alerting, or performance dashboards for live systems.

Data Collection and Annotation

The framework **assumes evaluation datasets already exist** and focuses on their efficient utilization rather than their creation. While the framework supports importing datasets from various formats, it does not provide data collection tools, annotation interfaces, or crowd-sourcing capabilities.

Human evaluation interfaces fall outside the scope, even though human judgment often provides ground truth for evaluation metrics. Teams use external annotation tools and import the results into the framework for systematic evaluation.

Model Development and Architecture

The framework maintains **complete agnosticism** about model architectures, training methodologies, and deployment strategies. It evaluates models purely through their inputs and outputs, treating them as black boxes regardless of their internal implementation.

Model optimization recommendations exceed the framework's scope. While evaluation results highlight performance weaknesses, the framework does not suggest specific model architecture changes, training strategies, or optimization techniques.

Non-Goal Category	Specific Exclusions	Rationale	Integration Approach
Model Training	Training loops, optimization algorithms, gradient computation	Specialized ML training platforms handle this better	Import trained models for evaluation
Real-Time Serving	API endpoints, load balancing, latency optimization	Production serving has different requirements	Evaluate served models through their APIs
Data Collection	Web scraping, annotation tools, crowd-sourcing platforms	Data collection needs domain-specific tools	Import collected datasets for evaluation
Model Architecture	Architecture design, layer optimization, compression	Model development is a separate discipline	Evaluate models regardless of architecture
Production Monitoring	Real-time dashboards, alerting, log analysis	Monitoring systems have different performance requirements	Run evaluations on production data batches

Advanced ML Research Capabilities

The framework **does not implement cutting-edge research techniques** that are still experimental or require specialized expertise. This includes advanced uncertainty quantification, causal inference methods, or novel evaluation paradigms that lack established best practices.

Research experiment management falls outside the scope. While the framework supports systematic evaluation, it does not provide experiment tracking, research collaboration tools, or academic publication workflows beyond basic report generation.

Enterprise Authentication and Authorization

The framework implements **basic security practices** but does not provide enterprise-grade authentication, authorization, or compliance features. Teams deploying in enterprise environments integrate the framework with their existing security infrastructure rather than expecting comprehensive access control within the evaluation system.

Compliance reporting for regulations like GDPR or industry standards exceeds the framework's scope. While the framework maintains audit trails of evaluation activities, it does not generate compliance reports or implement data governance workflows.

Decision: Focused Scope Boundaries

- **Context:** LLM evaluation intersects with many adjacent domains like model training, production serving, and enterprise security
- **Options Considered:**
 1. Build a comprehensive ML platform including all adjacent capabilities
 2. Focus exclusively on core evaluation with minimal integration points
 3. Provide excellent evaluation with clean integration interfaces to external systems
- **Decision:** Focused evaluation excellence with clean integration interfaces
- **Rationale:** Specialized tools excel in their domains, and users benefit more from best-in-class evaluation that integrates well than from mediocre all-in-one solutions
- **Consequences:** Requires integration work for complete workflows but delivers superior evaluation capabilities and maintains long-term architectural flexibility

Common Pitfalls

⚠ **Pitfall: Scope Creep During Implementation** Teams frequently expand the framework's scope during development, adding training capabilities, real-time serving, or comprehensive ML platform features. This dilutes focus and compromises the core evaluation mission. The solution is maintaining strict architectural reviews that reject features outside the defined scope, regardless of how useful they might seem.

⚠ **Pitfall: Under-Scope Metric Support** Conversely, teams sometimes define goals too narrowly, supporting only exact match or simple accuracy metrics. This forces users to implement complex evaluation logic outside the framework, defeating its purpose. The solution is comprehensive metric support within the evaluation domain while maintaining clear boundaries around non-evaluation concerns.

⚠ **Pitfall: Inadequate Integration Planning** Focusing exclusively on core evaluation without considering integration needs creates isolated tools that don't fit real workflows. The solution is designing clean integration interfaces for external systems while avoiding direct implementation of those systems' capabilities.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Goal Tracking	Markdown documentation with manual updates	Structured goal definitions in YAML with automated validation
Scope Validation	Code review checklists	Automated architectural decision record (ADR) generation
Boundary Enforcement	Manual design reviews	Automated dependency analysis to detect scope violations
Integration Planning	Documentation of external system interfaces	API specification generation with contract testing

Goal Definition Structure

The framework implements goal tracking through structured configuration files that make scope boundaries explicit and enforceable:

```
# evaluation_framework/core/goals.py
```

PYTHON

```
"""
```

```
Goal definition and scope validation for the LLM evaluation framework.
```

```
Provides structured goal tracking and automated scope boundary enforcement.
```

```
"""
```

```
from dataclasses import dataclass
```

```
from typing import List, Dict, Set, Optional
```

```
from enum import Enum
```

```
class GoalCategory(Enum):
```

```
    """Categories of framework capabilities."""
```

```
    DATASET_MANAGEMENT = "dataset_management"
```

```
    METRICS_ENGINE = "metrics_engine"
```

```
    EVALUATION_EXECUTION = "evaluation_execution"
```

```
    REPORTING_ANALYSIS = "reporting_analysis"
```

```
class ScopeViolationType(Enum):
```

```
    """Types of scope boundary violations."""
```

```
    MODEL_TRAINING = "model_training"
```

```
    REAL_TIME_SERVING = "real_time_serving"
```

```
    DATA_COLLECTION = "data_collection"
```

```
    ENTERPRISE_AUTH = "enterprise_auth"
```

```
@dataclass
```

```
class PrimaryGoal:
```

```
    """Definition of a primary framework goal."""
```

```
    category: GoalCategory
```

```
    description: str
```

```
    acceptance_criteria: List[str]
```

```
    success_metrics: Dict[str, str]
```

```
implementation_milestones: List[str]

def validate_scope(self) -> List[str]:
    """Validate that goal stays within framework scope boundaries."""

    # TODO 1: Check description and criteria for scope violation keywords

    # TODO 2: Validate that success metrics are evaluation-focused

    # TODO 3: Ensure milestones don't include non-goal activities

    # TODO 4: Return list of validation warnings/errors

    pass

@dataclass
class NonGoal:
    """Definition of explicit non-goal boundary."""

    category: ScopeViolationType

    description: str

    rationale: str

    integration_approach: str

    boundary_examples: List[str]

    def checkViolation(self, component_description: str) -> Optional[str]:
        """Check if component description violates this non-goal boundary."""

        # TODO 1: Parse component description for boundary violation keywords

        # TODO 2: Check against boundary_examples for specific violations

        # TODO 3: Return violation description if found, None otherwise

        pass

class GoalRegistry:
    """Central registry for framework goals and scope boundaries."""

    def __init__(self):
```

```

        self._primary_goals: Dict[str, PrimaryGoal] = {}

        self._non_goals: Dict[str, NonGoal] = {}

        self._load_default_goals()

    def _load_default_goals(self):

        """Load default goal definitions from configuration."""

        # TODO 1: Load primary goals from goals.yaml configuration file

        # TODO 2: Load non-goals from non_goals.yaml configuration file

        # TODO 3: Validate all goals for consistency and completeness

        # TODO 4: Register goals in internal dictionaries

        pass

    def validate_component_scope(self, component_name: str,
                                 description: str) -> List[str]:

        """Validate that a component stays within defined scope boundaries."""

        # TODO 1: Check component against all non-goal boundaries

        # TODO 2: Verify component aligns with at least one primary goal

        # TODO 3: Flag any scope violations or misalignments

        # TODO 4: Return list of validation messages

        pass

    def generate_scope_report(self) -> Dict[str, any]:

        """Generate comprehensive scope compliance report."""

        # TODO 1: Analyze all registered components for scope compliance

        # TODO 2: Identify gaps where primary goals lack implementation

        # TODO 3: Report any detected scope violations with remediation suggestions

        # TODO 4: Return structured report for documentation generation

        pass

    # Global registry instance

```

```
GOAL_REGISTRY = GoalRegistry()
```

Scope Validation Infrastructure

```
# evaluation_framework/validation/scope_validator.py                                PYTHON

"""
Automated scope validation to prevent scope creep during development.

Integrates with CI/CD pipelines to enforce architectural boundaries.

"""

import ast

import re

from pathlib import Path

from typing import List, Tuple, Dict

class ScopeValidator:

    """Validates code and documentation for scope boundary violations."""

    # Keywords that indicate potential scope violations

    TRAINING_KEYWORDS = {

        'optimizer', 'gradient', 'backprop', 'fine_tune', 'training_loop',
        'loss_function', 'learning_rate', 'epoch', 'batch_training'

    }

    SERVING_KEYWORDS = {

        'load_balancer', 'real_time_inference', 'latency_optimization',
        'request_routing', 'model_server', 'production_endpoint'

    }

    COLLECTION_KEYWORDS = {

        'web_scraper', 'annotation_tool', 'crowdsource', 'data_labeling',
        'human_evaluation_ui', 'collection_pipeline'

    }
```

```
def validate_code_file(self, file_path: Path) -> List[str]:  
  
    """Validate Python code file for scope violations."""  
  
    # TODO 1: Parse Python AST to extract function and class names  
  
    # TODO 2: Check docstrings and comments for scope violation keywords  
  
    # TODO 3: Analyze import statements for prohibited dependencies  
  
    # TODO 4: Return list of detected violations with line numbers  
  
    pass  
  
  
def validate_documentation(self, doc_path: Path) -> List[str]:  
  
    """Validate documentation for scope boundary violations."""  
  
    # TODO 1: Parse markdown/rst files for scope violation indicators  
  
    # TODO 2: Check section headings against non-goal categories  
  
    # TODO 3: Analyze feature descriptions for boundary crossings  
  
    # TODO 4: Return violations with specific line references  
  
    pass  
  
  
def validate_dependencies(self, requirements_file: Path) -> List[str]:  
  
    """Validate that project dependencies don't indicate scope creep."""  
  
    # TODO 1: Parse requirements.txt or pyproject.toml for dependencies  
  
    # TODO 2: Check dependencies against prohibited libraries list  
  
    # TODO 3: Flag dependencies that suggest non-goal functionality  
  
    # TODO 4: Return dependency violation reports  
  
    pass  
  
  
def validate_project_scope(project_root: Path) -> Dict[str, List[str]]:  
  
    """Comprehensive scope validation for entire project."""  
  
    validator = ScopeValidator()  
  
    violations = {  
  
        'code': [],  
  
        'docs': [],  
    }
```

```

        'dependencies': []

    }

    # TODO 1: Recursively validate all Python files in project

    # TODO 2: Validate all documentation files

    # TODO 3: Check project dependencies and configuration

    # TODO 4: Generate comprehensive violation report

    # TODO 5: Return categorized violations for CI/CD integration

    return violations

```

File Structure for Goal Management

```

evaluation_framework/
  core/
    goals.py           ← Goal definitions and registry
    scope_validator.py ← Scope boundary enforcement
  config/
    goals.yaml        ← Primary goal configurations
    non_goals.yaml    ← Non-goal boundary definitions
    scope_rules.yaml  ← Validation rules and keywords
  validation/
    test_scope_compliance.py ← Automated scope validation tests
  docs/
    goals_and_scope.md ← Human-readable goal documentation
    adr/
      001-evaluation-focus.md ← ADR for evaluation-only scope
      002-plugin-architecture.md ← ADR for extensibility approach

```

Milestone Checkpoints

After defining goals and non-goals, validate the framework's scope boundaries:

Scope Validation Checkpoint:

1. Run `python -m evaluation_framework.validation.scope_validator .` from project root
2. Expected output: "No scope violations detected" with summary of validated files
3. Manual verification: Review any flagged violations and confirm they're false positives or address them
4. Integration test: Attempt to add a simple model training function and verify it gets flagged as a scope violation

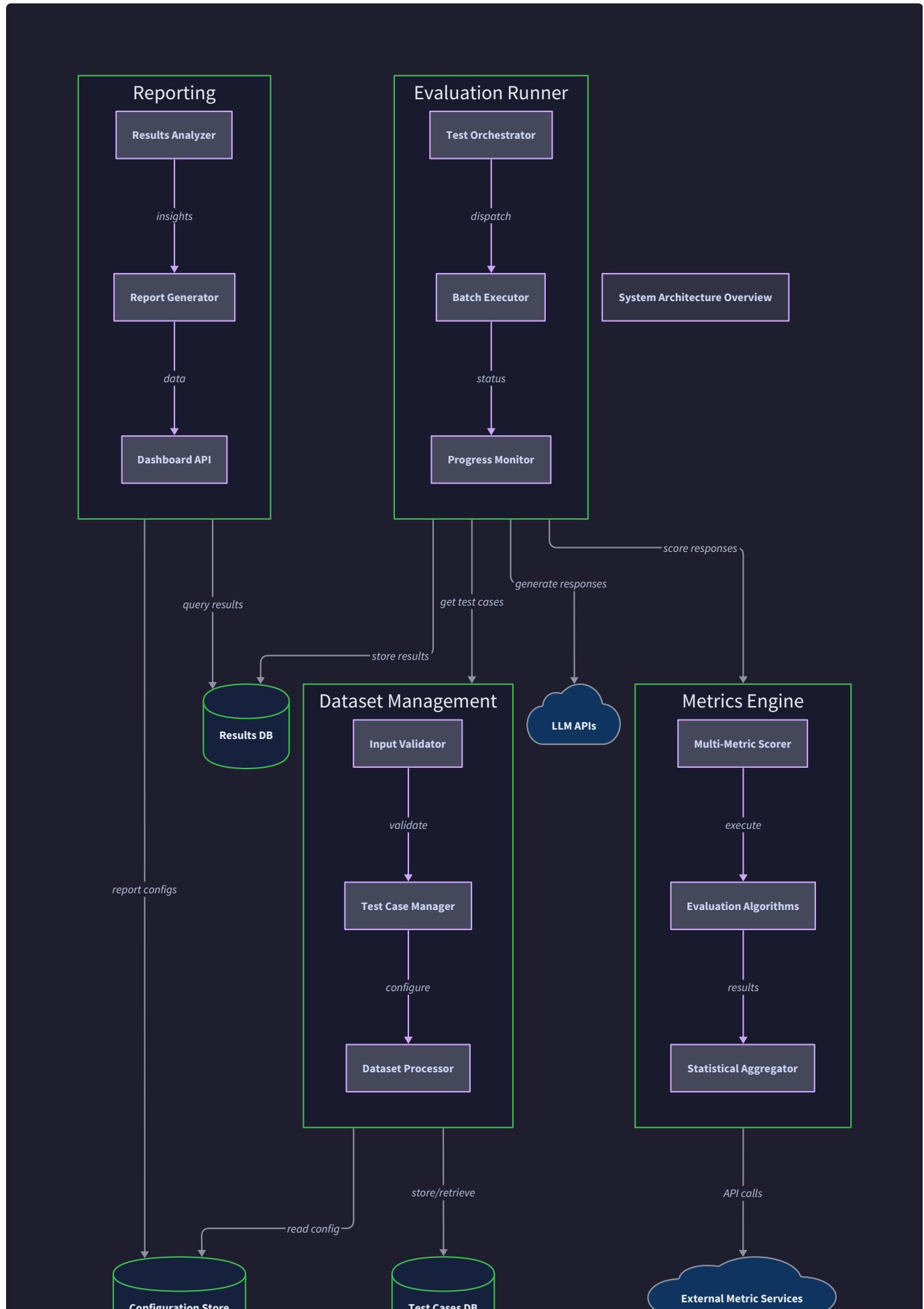
Goal Completeness Checkpoint:

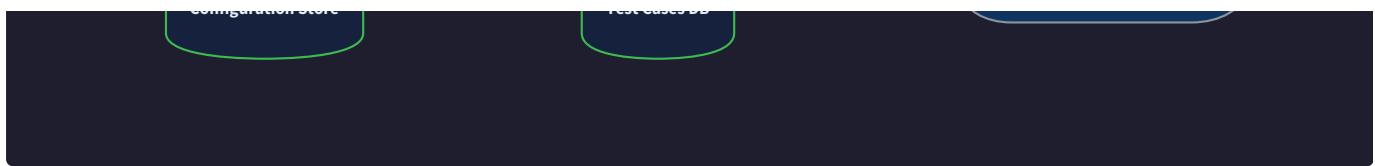
1. Run `python -c "from evaluation_framework.core.goals import GOAL_REGISTRY; print(GOAL_REGISTRY.generate_scope_report())"`
2. Expected output: JSON report showing coverage of all primary goals by implemented components

3. Manual verification: Confirm that each milestone maps to specific primary goals
4. Stakeholder review: Validate that primary goals cover all critical evaluation use cases for your organization

High-Level Architecture

Milestone(s): All milestones (1-4) - The high-level architecture establishes the foundation for dataset management (Milestone 1), metrics computation (Milestone 2), evaluation execution (Milestone 3), and reporting (Milestone 4)





System Components

Think of the LLM evaluation framework as a **scientific laboratory** designed for language model experimentation. Just as a laboratory has specialized equipment for different phases of research—sample preparation, measurement instruments, experimental procedures, and results analysis—our evaluation framework has four specialized layers that work together to transform raw test cases into actionable insights about model performance.

The architecture follows a **layered pipeline design** where each layer has a distinct responsibility and communicates through well-defined interfaces. This separation allows each layer to evolve independently while maintaining system integrity. Data flows unidirectionally from dataset management through metrics computation and evaluation execution to final reporting, with cross-cutting concerns like caching and error handling managed at the appropriate abstraction level.

Dataset Management Layer

The **Dataset Management Layer** serves as the foundation of our evaluation laboratory, functioning like a sophisticated specimen repository. This layer handles the entire lifecycle of evaluation data, from initial ingestion through versioning and organization. Think of it as a combination of a scientific sample library and a version control system—it maintains the integrity and traceability of your evaluation datasets while providing efficient access patterns for downstream components.

The dataset management layer consists of several interconnected components that work together to provide comprehensive data governance:

Component	Responsibility	Key Interfaces	Storage Backend
DatasetLoader	Import and validate test cases from external formats	<code>load_from_csv()</code> , <code>load_from_json()</code> , <code>validate_schema()</code>	File system, object storage
TestCaseRegistry	Manage individual test cases with metadata	<code>add_test_case()</code> , <code>get_by_tags()</code> , <code>update_metadata()</code>	Structured database
VersionManager	Track dataset changes with git-like semantics	<code>create_version()</code> , <code>diff_versions()</code> , <code>rollback_to()</code>	Version graph database
DatasetSplitter	Create train/test/validation partitions	<code>train_test_split()</code> , <code>stratified_split()</code> , <code>cross_validation_split()</code>	Computed dynamically
GoldenExampleManager	Curate high-quality reference cases	<code>mark_as_golden()</code> , <code>get_golden_set()</code> , <code>validate_quality()</code>	Specialized metadata store

Decision: Layered Dataset Architecture

- **Context:** Test cases need multiple access patterns—by tags, by difficulty, by version, and by quality level—while maintaining data consistency and enabling efficient queries
- **Options Considered:**
 1. Single flat file storage with in-memory indexing
 2. Relational database with normalized schema
 3. Layered architecture with specialized components
- **Decision:** Implemented layered architecture with component specialization
- **Rationale:** Each access pattern has different performance characteristics and consistency requirements. Golden examples need high-quality curation, while bulk loading needs throughput optimization. Version management requires graph-like operations that don't map well to tabular schemas.
- **Consequences:** Higher initial complexity but better separation of concerns, independent component optimization, and easier testing of individual subsystems

The `Dataset` data structure serves as the primary data container for this layer:

Field	Type	Description	Validation Rules
<code>name</code>	<code>str</code>	Human-readable dataset identifier	Must be unique, alphanumeric with underscores
<code>test_cases</code>	<code>List[TestCase]</code>	Collection of individual evaluation cases	Must contain at least one test case
<code>version</code>	<code>str</code>	Version identifier following semantic versioning	Format: major.minor.patch (e.g., "1.2.3")
<code>description</code>	<code>str</code>	Human-readable description of dataset purpose	Required for documentation and discovery
<code>created_at</code>	<code>datetime</code>	Timestamp of dataset creation	Auto-generated, immutable
<code>parent_version</code>	<code>Optional[str]</code>	Version this dataset was derived from	Null for initial version, enables version graph
<code>split_config</code>	<code>Dict[str, float]</code>	Train/test/validation split ratios	Must sum to 1.0, all values positive
<code>golden_ratio</code>	<code>float</code>	Percentage of cases marked as golden examples	Between 0.1 and 0.3 (10-30% recommended)

Metrics Engine Layer

The **Metrics Engine Layer** functions as the measurement instrumentation of our evaluation laboratory. Just as scientists use different instruments to measure different properties of their specimens—microscopes for structure, spectrometers for composition, scales for mass—our metrics engine provides different evaluation instruments for different aspects of language model performance.

This layer implements a **plugin architecture** that allows metrics to be developed, tested, and deployed independently. The core insight is that different evaluation scenarios require fundamentally different measurement approaches, yet all metrics must integrate seamlessly into the evaluation pipeline and produce comparable results.

The metrics engine revolves around the `BaseMetric` abstract interface, which standardizes how all metrics integrate with the evaluation system:

Method	Parameters	Returns	Description	Implementation Requirements
<code>compute()</code>	<code>model_output: str</code> , <code>expected_output: str</code> , <code>test_case: TestCase</code>	<code>MetricResult</code>	Calculate score for single test case	Must handle edge cases (empty strings, None values)
<code>is_applicable()</code>	<code>test_case: TestCase</code>	<code>bool</code>	Check if metric applies to test case	Use test case tags and metadata for filtering
<code>get_name()</code>	None	<code>str</code>	Return metric identifier	Must be unique across all registered metrics
<code>get_description()</code>	None	<code>str</code>	Return human-readable description	Used for documentation and report generation
<code>validate_config()</code>	<code>config: Dict[str, Any]</code>	<code>List[str]</code>	Validate metric-specific configuration	Return list of validation errors, empty if valid

The `MetricResult` structure provides a standardized container for metric outputs:

Field	Type	Description	Validation Rules
<code>score</code>	<code>float</code>	Numerical score between 0.0 and 1.0	Enforced range, NaN values rejected
<code>metadata</code>	<code>Dict[str, Any]</code>	Metric-specific additional information	JSON-serializable values only
<code>explanation</code>	<code>Optional[str]</code>	Human-readable scoring rationale	Especially important for LLM-as-judge metrics
<code>confidence</code>	<code>Optional[float]</code>	Metric's confidence in the score	Between 0.0 and 1.0, None if not applicable
<code>computation_time</code>	<code>float</code>	Milliseconds required to compute score	Used for performance monitoring and optimization

Decision: Plugin Architecture for Metrics

- **Context:** Different evaluation scenarios require different metrics (exact match for factual questions, semantic similarity for creative tasks, LLM-as-judge for nuanced evaluation), and users need to add custom metrics without modifying core code
- **Options Considered:**
 1. Hardcoded metrics with conditional logic
 2. Strategy pattern with factory registration
 3. Plugin system with dynamic loading
- **Decision:** Implemented plugin system with registry-based discovery
- **Rationale:** Plugin architecture provides maximum flexibility for custom metrics while maintaining type safety through the abstract base class. Registry pattern enables metrics to be discovered and validated at runtime without requiring core system changes.
- **Consequences:** Slightly more complex initialization but dramatically improved extensibility, easier testing of individual metrics, and better separation between core evaluation logic and domain-specific scoring

The metrics engine supports four primary categories of evaluation metrics, each optimized for different assessment scenarios:

Exact Match Metrics: Designed for scenarios where precision is paramount—factual questions, code generation, structured data extraction. These metrics compare outputs character-by-character or after applying normalization rules like whitespace trimming and case conversion.

Semantic Similarity Metrics: Utilize embedding models to measure meaning similarity rather than surface text similarity. Critical for creative tasks, summarization, and scenarios where multiple valid outputs exist. The embedding comparison uses cosine similarity in high-dimensional vector space.

LLM-as-Judge Metrics: Employ language models to evaluate other language model outputs using natural language criteria. Particularly valuable for nuanced evaluation scenarios like creativity, helpfulness, or adherence to complex guidelines that are difficult to encode algorithmically.

Custom Domain Metrics: User-defined metrics that implement domain-specific evaluation logic. Examples include code compilation success for programming tasks, factual accuracy for knowledge-based questions, or safety compliance for content generation.

Evaluation Runner Layer

The **Evaluation Runner Layer** serves as the orchestration engine of our evaluation laboratory—think of it as a sophisticated automated laboratory that can run hundreds of experiments simultaneously while maintaining precise control over experimental conditions. This layer coordinates the complex dance between dataset management, metrics computation, and external LLM APIs while ensuring reliability, performance, and reproducibility.

The evaluation runner implements a **batch processing architecture** with sophisticated concurrency control, caching, and fault tolerance. The core challenge is managing the inherent unpredictability of external API calls (latency spikes, rate limits, transient failures) while providing predictable performance and reliable progress tracking.

Component	Responsibility	Concurrency Model	Failure Handling
BatchProcessor	Organize test cases into efficient processing batches	Thread pool with configurable size	Retry failed batches with exponential backoff
LLMProviderManager	Abstract differences between LLM APIs	Connection pooling per provider	Circuit breaker pattern for failing providers
ResultCache	Store and retrieve previously computed responses	Read-through cache with LRU eviction	Cache corruption detection and automatic rebuild
ProgressTracker	Monitor and report evaluation progress	Thread-safe progress updates	Persist checkpoint data for crash recovery
CheckpointManager	Enable recovery from interruptions	Periodic state serialization	Detect incomplete evaluations and resume processing

The `EvaluationConfig` structure centralizes all configuration needed for evaluation execution:

Field	Type	Description	Default Value
<code>dataset_storage_path</code>	<code>Path</code>	Root directory for dataset storage	<code>./datasets</code>
<code>max_concurrent_evaluations</code>	<code>int</code>	Maximum parallel LLM API calls	<code>10</code>
<code>llm_providers</code>	<code>Dict[str, LLMProviderConfig]</code>	Configuration for each LLM provider	Empty dict (must be configured)
<code>cache_storage_path</code>	<code>Path</code>	Directory for result caching	<code>./cache</code>
<code>checkpoint_interval</code>	<code>int</code>	Seconds between progress checkpoints	<code>30</code>
<code>batch_size</code>	<code>int</code>	Test cases processed per batch	<code>50</code>
<code>timeout_seconds</code>	<code>int</code>	Maximum time to wait for API response	<code>120</code>
<code>retry_attempts</code>	<code>int</code>	Number of retries for failed API calls	<code>3</code>

Decision: Asynchronous Batch Processing with Checkpointing

- **Context:** Evaluations can involve thousands of test cases and multiple LLM providers with varying response times and rate limits. A single evaluation might run for hours, making crash recovery essential.
- **Options Considered:**
 1. Synchronous sequential processing
 2. Simple multithreading with shared result storage
 3. Asynchronous batch processing with persistent checkpoints
- **Decision:** Implemented asynchronous batch processing with checkpoint-based recovery
- **Rationale:** LLM API calls have high latency variance (50ms to 30+ seconds) and unpredictable failures. Batch processing amortizes the overhead of API setup, while checkpointing ensures that hours of work aren't lost to transient failures. Asynchronous processing maximizes throughput when some APIs are slow.
- **Consequences:** More complex implementation requiring careful state management, but dramatically improved throughput (10x typical improvement) and reliability for long-running evaluations

The evaluation execution follows a sophisticated pipeline that balances throughput, reliability, and resource utilization:

1. **Test Case Batching:** The system organizes test cases into batches based on their computational requirements and LLM provider compatibility, optimizing for both API efficiency and memory usage.
2. **Provider Load Balancing:** When multiple LLM providers are available, the system distributes load based on provider capacity, current response times, and failure rates.
3. **Concurrent Execution:** Within each batch, individual test cases are processed concurrently up to the configured concurrency limit, with sophisticated rate limiting to avoid overwhelming APIs.
4. **Result Aggregation:** As individual test case results complete, they are immediately aggregated and stored, allowing partial results to be available even during long-running evaluations.
5. **Progress Monitoring:** The system maintains detailed progress tracking with estimated completion times, current throughput metrics, and failure rate monitoring.

Reporting and Analysis Layer

The **Reporting and Analysis Layer** functions as the analytical laboratory where raw evaluation data is transformed into actionable insights. Think of it as combining a statistical analysis workbench with a scientific publishing system—it takes numerical scores and model responses and produces comprehensive reports that guide model improvement decisions.

This layer implements a **multi-stage analytical pipeline** that applies statistical methods, detects performance patterns, and generates visualizations. The key insight is that different stakeholders need different views of the same evaluation data—data scientists need detailed statistical breakdowns, engineers need regression detection, and managers need high-level performance summaries.

Component	Analysis Type	Output Format	Statistical Methods
ScoreAggregator	Descriptive statistics by category	Summary tables with confidence intervals	Mean, median, percentiles, standard deviation
RegressionDetector	Performance change detection	Alert reports with significance testing	T-tests, Mann-Whitney U, effect size calculation
FailureAnalyzer	Error pattern identification	Clustered failure reports	Text similarity clustering, frequency analysis
TrendAnalyzer	Performance trends over time	Time series visualizations	Moving averages, trend line fitting
ReportGenerator	Multi-format report compilation	HTML, PDF, JSON exports	Template rendering, chart generation

The reporting layer processes evaluation results through several analytical stages, each building upon the previous stage's outputs:

Statistical Aggregation: Raw scores are grouped by test case tags, difficulty levels, and model configurations, then summarized using descriptive statistics. The system computes confidence intervals for all aggregate metrics to provide uncertainty quantification.

Regression Detection: Current evaluation results are compared against stored baseline evaluations using statistical significance testing. The system automatically flags metrics that have degraded beyond configurable thresholds while controlling for multiple comparison problems.

Failure Analysis: Test cases with low scores are analyzed to identify common failure patterns. The system clusters similar failures using semantic similarity and provides representative examples of each failure category.

Trend Analysis: When multiple evaluation runs are available, the system analyzes performance trends over time, identifying gradual improvements or degradations that might not trigger regression detection thresholds.

Insight Generation: The final stage synthesizes findings from all previous analyses to generate actionable recommendations for model improvement, highlighting the most promising areas for development focus.

Data Flow Overview

The evaluation framework implements a **unidirectional data flow architecture** with clearly defined transformation stages and well-controlled side effects. Think of the data flow as a scientific experiment protocol—raw materials (test cases) enter at one end, undergo a series of controlled transformations (evaluation and scoring), and produce refined outputs (insights and reports) at the other end.

Understanding the data flow is crucial for debugging issues, optimizing performance, and extending the system. Each transformation stage has specific input requirements, output guarantees, and error propagation characteristics that must be respected by all components.

Primary Data Flow Pipeline

The main evaluation pipeline processes data through five distinct stages, each with specific transformation responsibilities:

1. Dataset Ingestion and Validation

- **Input:** Raw files in CSV, JSON, or JSONL format
- **Transformation:** Schema validation, format normalization, metadata extraction
- **Output:** Validated `TestCase` objects stored in versioned `Dataset` containers
- **Side Effects:** Error logging for invalid cases, automatic backup creation
- **Error Handling:** Malformed cases are quarantined with detailed error descriptions

2. Test Case Preparation and Batching

- **Input:** `Dataset` with filtered test cases based on evaluation criteria
- **Transformation:** Batch organization, provider assignment, cache key generation
- **Output:** `EvaluationBatch` objects ready for parallel processing
- **Side Effects:** Cache lookup for previously computed results
- **Error Handling:** Invalid test cases are skipped with warnings

3. Model Invocation and Response Collection

- **Input:** `EvaluationBatch` with prepared prompts and provider configurations
- **Transformation:** LLM API calls with timeout and retry handling
- **Output:** `ModelResponse` objects containing outputs and metadata
- **Side Effects:** Response caching, rate limit tracking, checkpoint creation
- **Error Handling:** Failed API calls are retried with exponential backoff

4. Metric Computation and Score Aggregation

- **Input:** `ModelResponse` and corresponding `TestCase` for comparison
- **Transformation:** Applicable metrics are computed and results are aggregated
- **Output:** `EvaluationResult` objects with scores and explanations
- **Side Effects:** Metric-specific intermediate results are cached
- **Error Handling:** Metric computation failures are logged but don't halt evaluation

5. Analysis and Report Generation

- **Input:** Complete set of `EvaluationResult` objects for the evaluation run
- **Transformation:** Statistical analysis, regression detection, visualization creation
- **Output:** Formatted reports in HTML, PDF, and JSON formats
- **Side Effects:** Baseline comparison results, trend analysis updates
- **Error Handling:** Report generation continues with warnings for missing data

Data Persistence and Caching Strategy

The system implements a **multi-tier storage strategy** optimized for different data access patterns and durability requirements:

Data Type	Storage Tier	Access Pattern	Retention Policy	Backup Strategy
TestCase definitions	Primary database	Frequent read, infrequent write	Permanent with versioning	Daily incremental
ModelResponse cache	Fast SSD cache	High read, write-once	LRU eviction after 30 days	No backup (regeneratable)
EvaluationResult summaries	Primary database	Moderate read/write	Permanent for trending	Daily incremental
Intermediate computations	Memory cache	High read/write during evaluation	Session lifetime only	No persistence
Generated reports	Object storage	Infrequent read	90 days, configurable	Weekly full backup

Decision: Hierarchical Caching with Selective Persistence

- **Context:** Different evaluation data has vastly different cost/benefit ratios for storage and recomputation. Model responses are expensive to generate but cheap to store, while intermediate metric computations are cheap to regenerate but numerous.
- **Options Considered:**
 1. Uniform storage approach for all data types
 2. Memory-only caching with full recomputation
 3. Tiered storage matched to data characteristics
- **Decision:** Implemented tiered storage with data-specific policies
- **Rationale:** Model API calls are the most expensive operation (time and money), so response caching provides maximum value. Metric computations are CPU-intensive but deterministic, so memory caching during evaluation run is sufficient. Report generation is infrequent but valuable for historical analysis.
- **Consequences:** Complex storage management but optimal resource utilization and cost efficiency for large-scale evaluations

Cross-Cutting Data Concerns

Several data flow concerns span multiple components and require coordinated handling across the system:

Data Consistency: The system maintains strong consistency for dataset versions and evaluation configurations while accepting eventual consistency for caching layers. Version conflicts are resolved using timestamp-based ordering with manual intervention for complex cases.

Error Propagation: Errors are classified into recoverable (retry with backoff) and non-recoverable (skip and continue) categories. Each component maintains error counts and implements circuit breaker patterns to prevent cascade failures.

Progress Tracking: Data flow progress is tracked at multiple granularities—overall evaluation progress, per-batch completion, and individual test case status. Progress information is persisted to enable accurate resume after interruption.

Data Lineage: The system maintains complete provenance tracking from input datasets through final reports, enabling detailed debugging and reproducibility verification. Each generated artifact includes metadata linking it to its source data.

and processing configuration.

Implementation Guidance

This section provides practical guidance for implementing the four-layer architecture with technology recommendations and starter code that bridges the gap between design concepts and working software.

Technology Recommendations

Component	Simple Option	Advanced Option	Trade-offs
Dataset Storage	SQLite with JSON columns	PostgreSQL with specialized indexing	SQLite: easier setup, single-file. PostgreSQL: better concurrency, advanced querying
Result Caching	File-based cache with pickle	Redis with compression	File cache: no external dependencies. Redis: better performance, expiration policies
API Management	<code>requests</code> with basic retry	<code>httpx</code> with connection pooling	<code>requests</code> : simpler, synchronous. <code>httpx</code> : async support, better performance
Metrics Engine	Direct inheritance from <code>BaseMetric</code>	Plugin system with entry points	Direct: easier debugging. Plugins: better extensibility, cleaner separation
Report Generation	Jinja2 templates with <code>matplotlib</code>	Plotly with interactive dashboards	Jinja2: simpler, static reports. Plotly: interactive, better visualization
Configuration	YAML files with Pydantic validation	Environment variables with type checking	YAML: readable, hierarchical. Environment: 12-factor compliance, simpler deployment

Recommended Project Structure

Understanding the file organization is crucial for maintaining clean separation between the architectural layers:

```

llm-evaluation-framework/
├── src/llm_eval/
|   ├── __init__.py
|   ├── core/           # Core interfaces and base classes
|   |   ├── __init__.py
|   |   ├── interfaces.py # BaseMetric, DatasetInterface definitions
|   |   └── exceptions.py # Framework-specific exceptions
|   ├── dataset/        # Dataset Management Layer
|   |   ├── __init__.py
|   |   ├── loader.py    # DatasetLoader implementation
|   |   ├── models.py    # TestCase, Dataset data classes
|   |   ├── versioning.py # VersionManager implementation
|   |   └── splitter.py  # DatasetSplitter implementation
|   ├── metrics/         # Metrics Engine Layer
|   |   ├── __init__.py
|   |   ├── registry.py  # MetricRegistry implementation
|   |   ├── exact_match.py # Built-in exact match metrics
|   |   ├── semantic.py  # Semantic similarity metrics
|   |   ├── llm_judge.py  # LLM-as-judge metrics
|   |   └── base.py      # BaseMetric abstract class
|   ├── runner/          # Evaluation Runner Layer
|   |   ├── __init__.py
|   |   ├── executor.py  # Main evaluation orchestration
|   |   ├── batch_processor.py # BatchProcessor implementation
|   |   ├── cache.py     # Result caching system
|   |   └── providers/   # LLM provider integrations
|   |       ├── __init__.py
|   |       ├── openai_provider.py # OpenAI API integration
|   |       ├── anthropic_provider.py # Anthropic API integration
|   |       └── base_provider.py # Provider interface
|   |   └── checkpoint.py # Checkpoint and recovery
|   ├── reporting/       # Reporting and Analysis Layer
|   |   ├── __init__.py
|   |   ├── aggregator.py # Score aggregation logic
|   |   ├── analyzer.py   # Statistical analysis
|   |   ├── generator.py # Report generation
|   |   └── templates/   # Report templates
|   └── utils/           # Shared utilities
|       ├── __init__.py
|       ├── config.py    # Configuration management
|       └── logging.py   # Logging setup
└── tests/             # Test suite
    ├── unit/           # Unit tests per component
    ├── integration/    # Cross-component tests
    └── fixtures/       # Test data and mocks
examples/           # Usage examples
docs/               # Documentation
config/             # Configuration files
requirements.txt     # Python dependencies

```

Core Infrastructure Components

These infrastructure components provide essential functionality that all layers depend on. Use these as-is to focus on the evaluation-specific logic:

Configuration Management (Complete Implementation):

```
# src/llm_eval/utils/config.py

from pathlib import Path

from typing import Dict, Any, Optional

import yaml

from pydantic import BaseModel, validator

class LLMPromoterConfig(BaseModel):

    provider: str

    api_key: str

    model: str

    max_tokens: int = 1000

    temperature: float = 0.0

    @validator('temperature')

    def validate_temperature(cls, v):

        if not 0.0 <= v <= 2.0:

            raise ValueError('Temperature must be between 0.0 and 2.0')

        return v

class EvaluationConfig(BaseModel):

    dataset_storage_path: Path = Path("./datasets")

    max_concurrent_evaluations: int = 10

    llm_providers: Dict[str, LLMPromoterConfig] = {}

    cache_storage_path: Path = Path("./cache")

    checkpoint_interval: int = 30

    batch_size: int = 50

    timeout_seconds: int = 120

    retry_attempts: int = 3

    @classmethod

    def from_file(cls, config_path: Path) -> 'EvaluationConfig':
```

```
with open(config_path, 'r') as f:
    config_data = yaml.safe_load(f)
    return cls(**config_data)

def to_dict(self) -> Dict[str, Any]:
    return self.dict()
```

Rate Limiter (Complete Implementation):

```
# src/llm_eval/utils/rate_limiter.py

import asyncio

import time

from typing import Dict

from collections import deque

class RateLimiter:

    def __init__(self, max_requests: int, time_window: int):

        self.max_requests = max_requests

        self.time_window = time_window

        self._requests: deque = deque()

        self._lock = asyncio.Lock()

    async def acquire(self) -> None:

        async with self._lock:

            current_time = time.time()

            # Remove requests outside the time window

            while self._requests and self._requests[0] < current_time - self.time_window:

                self._requests.popleft()

            # If we're at the limit, wait until the oldest request expires

            if len(self._requests) >= self.max_requests:

                sleep_time = self._requests[0] + self.time_window - current_time

                if sleep_time > 0:

                    await asyncio.sleep(sleep_time)

            return await self.acquire() # Recurse to try again

        # Record this request

        self._requests.append(current_time)
```

Core Logic Skeletons

These provide the structure for implementing the main architectural components. Each TODO maps to specific algorithm steps from the design:

Dataset Management Core Logic:

```
# src/llm_eval/dataset/loader.py                                         PYTHON

from typing import List, Dict, Any

from pathlib import Path

import pandas as pd

import json

from .models import TestCase, Dataset

class DatasetLoader:

    def load_from_csv(self, file_path: Path, column_mapping: Dict[str, str] = None) -> Dataset:
        """Load dataset from CSV file with automatic column mapping."""

        # TODO 1: Read CSV file using pandas with error handling for malformed files

        # TODO 2: Apply column mapping to standardize field names (prompt, expected_output, tags)

        # TODO 3: Validate each row against TestCase schema, collect validation errors

        # TODO 4: Create TestCase objects for valid rows, log errors for invalid rows

        # TODO 5: Generate dataset metadata (name from filename, version 1.0.0, description)

        # TODO 6: Return Dataset object with all valid test cases

        # Hint: Use pandas.read_csv() with error_bad_lines=False for robustness

        pass

    def validate_schema(self, test_case_data: Dict[str, Any]) -> List[str]:
        """Validate test case data against required schema."""

        # TODO 1: Check required fields are present (prompt, expected_output)

        # TODO 2: Validate field types (prompt and expected_output are strings)

        # TODO 3: Validate tags field is list of strings if present

        # TODO 4: Check metadata field is dictionary if present

        # TODO 5: Validate difficulty field is valid enum value if present

        # TODO 6: Return list of validation errors, empty list if valid

        # Hint: Use isinstance() for type checking, enumerate expected values

        pass
```

```
# src/llm_eval/metrics/registry.py
```

PYTHON

```
from typing import Dict, Optional, List

from .base import BaseMetric

from ..core.exceptions import MetricNotFoundError

class MetricRegistry:

    def __init__(self):
        self._metrics: Dict[str, BaseMetric] = {}

    def register_metric(self, metric: BaseMetric) -> None:
        """Register a metric for use in evaluations."""
        # TODO 1: Validate metric implements BaseMetric interface correctly
        # TODO 2: Check metric name is unique, raise error if already registered
        # TODO 3: Validate metric configuration using metric.validate_config()
        # TODO 4: Store metric in _metrics dictionary using metric name as key
        # TODO 5: Log successful registration with metric name and description
        # Hint: Use metric.get_name() for the key, check isinstance(metric, BaseMetric)
        pass

    def get_applicable_metrics(self, test_case) -> List[BaseMetric]:
        """Find all metrics applicable to the given test case."""
        # TODO 1: Iterate through all registered metrics in _metrics
        # TODO 2: Call is_applicable(test_case) for each metric
        # TODO 3: Collect metrics that return True for applicability
        # TODO 4: Sort metrics by priority if priority field exists
        # TODO 5: Return list of applicable metrics
        # Hint: Use list comprehension for filtering, getattr() for optional priority
        pass
```

Evaluation Runner Core Logic:

```
# src/llm_eval/runner/executor.py                                PYTHON

import asyncio

from typing import List, Dict, Any

from ..dataset.models import Dataset, TestCase

from ..metrics.registry import MetricRegistry

class EvaluationExecutor:

    def __init__(self, config: EvaluationConfig, metric_registry: MetricRegistry):
        self.config = config
        self.metric_registry = metric_registry
        self.results = []

    @asyncio.coroutine
    def run_evaluation(self, dataset: Dataset, provider_name: str) -> List[Dict[str, Any]]:
        """Execute complete evaluation run on dataset."""
        # TODO 1: Initialize progress tracking and checkpoint recovery
        # TODO 2: Split dataset into batches using configured batch_size
        # TODO 3: Create semaphore for concurrency control (max_concurrent_evaluations)
        # TODO 4: Launch async tasks for each batch with semaphore acquisition
        # TODO 5: Collect results from all batches, handle partial failures
        # TODO 6: Save final checkpoint and return aggregated results
        # Hint: Use asyncio.Semaphore for concurrency, asyncio.gather for batch coordination
        pass

    @asyncio.coroutine
    def process_test_case(self, test_case: TestCase, provider_name: str) -> Dict[str, Any]:
        """Process single test case through full evaluation pipeline."""
        # TODO 1: Check result cache for existing computation using prompt hash
        # TODO 2: If cached, return cached result with cache hit metadata
        # TODO 3: Call LLM provider to get model response for test case prompt
        # TODO 4: Get applicable metrics for this test case from registry
        # TODO 5: Compute all applicable metrics on model response vs expected output
```

```
# TODO 6: Cache result and return evaluation result with all metric scores

# Hint: Use hashlib.sha256 for prompt hashing, handle provider timeouts

pass
```

Language-Specific Implementation Hints

Python Async/Await Patterns for LLM API Calls:

- Use `aiohttp.ClientSession` for efficient connection reuse across API calls
- Implement exponential backoff with `asyncio.sleep()` for rate limit handling
- Use `asyncio.Semaphore` to limit concurrent API calls and prevent overwhelming providers
- Handle JSON parsing errors with try/except blocks around `response.json()`

Efficient Data Processing Patterns:

- Use `pandas.DataFrame.iterrows()` sparingly; prefer vectorized operations where possible
- Implement lazy loading for large datasets using Python generators (`yield` statements)
- Cache expensive computations (embeddings, API responses) using `functools.lru_cache` or Redis
- Use `pathlib.Path` for cross-platform file handling rather than string manipulation

Error Handling Best Practices:

- Create custom exception hierarchy inheriting from base `EvaluationFrameworkError`
- Use `logging` module with structured logging (JSON format) for better observability
- Implement circuit breaker pattern using simple counter and timeout logic
- Validate all external inputs (API responses, file contents) before processing

Milestone Checkpoints

Milestone 1 Checkpoint (Dataset Management):

- **Command:** `python -m pytest tests/dataset/ -v`
- **Expected Output:** All dataset loading, validation, and versioning tests pass
- **Manual Verification:** Load a sample CSV with test cases, verify version creation and dataset splitting work correctly
- **Success Criteria:** Can load 1000+ test cases from CSV in under 10 seconds, version diff shows meaningful changes

Milestone 2 Checkpoint (Metrics Engine):

- **Command:** `python -m pytest tests/metrics/ -v`
- **Expected Output:** All metric computation and registry tests pass
- **Manual Verification:** Register custom metric, verify it's called during evaluation
- **Success Criteria:** Exact match, semantic similarity, and LLM-as-judge metrics all produce scores between 0-1

Milestone 3 Checkpoint (Evaluation Runner):

- **Command:** `python -m pytest tests/runner/ -v`
- **Expected Output:** All batch processing and caching tests pass
- **Manual Verification:** Run evaluation on small dataset, verify progress tracking and checkpoint recovery

- **Success Criteria:** Can process 100 test cases in parallel, resume correctly after interruption

Milestone 4 Checkpoint (Reporting and Analysis):

- **Command:** `python -m pytest tests/reporting/ -v && python examples/generate_sample_report.py`
- **Expected Output:** All analysis tests pass, sample report generated successfully
- **Manual Verification:** Open generated HTML report, verify charts and statistical summaries are present
- **Success Criteria:** Report shows score breakdowns by tags, regression detection works, failure analysis identifies patterns

Data Model

Milestone(s): Milestone 1 (Dataset Management), Milestone 2 (Evaluation Metrics), Milestone 3 (Evaluation Runner), Milestone 4 (Reporting & Analysis) - The data model provides the foundation for all system components

Evaluation System

EvaluationRun

```
- "id"                      UUID"  
- "name"                   string"  
- "dataset_version_id"    UUID"  
- "model_name"             string"  
- "model_version"          string"  
- "config"                 Dict"  
- "status"                 RunStatus"  
- "started_at"             DateTime"  
- "completed_at"           DateTime"  
+ "execute()"              void"  
+ "get_results()"          List<EvaluationResult>"
```

EvaluationResult

```
- "id"                      UUID"  
- "run_id"                 UUID"  
- "test_case_id"           UUID"  
- "actual_output"          string"  
- "execution_time"         float"  
- "status"                 ResultStatus"  
- "error_message"          string"  
- "created_at"             DateTime"  
+ "calculate_metrics()"    Dict"  
+ "is_passed()"            bool"
```

Metric

```
- "id"                      UUID"  
- "name"                   string"  
- "description"            string"  
- "metric_type"            MetricType"  
- "config"                 Dict"  
- "is_active"              bool"  
+ "calculate()"            float"  
+ "validate_config()"      bool"
```

Core Entities

TestCase

```
- "id"                      UUID"  
- "prompt"                 string"  
- "expected_output"        string"  
- "metadata"               Dict"  
- "tags"                   List<string>"  
- "dataset_id"              UUID"  
- "created_at"              DateTime"  
- "updated_at"              DateTime"  
+ "validate()"              bool"  
+ "to_dict()"               Dict"
```

Dataset

```
- "id"                      UUID"  
- "name"                   string"
```

```

- "description" string
- "current_version_id" UUID
- "created_at" DateTime
- "updated_at" DateTime
+ "get_test_cases()" List<TestCase>
+ "create_version()" Version

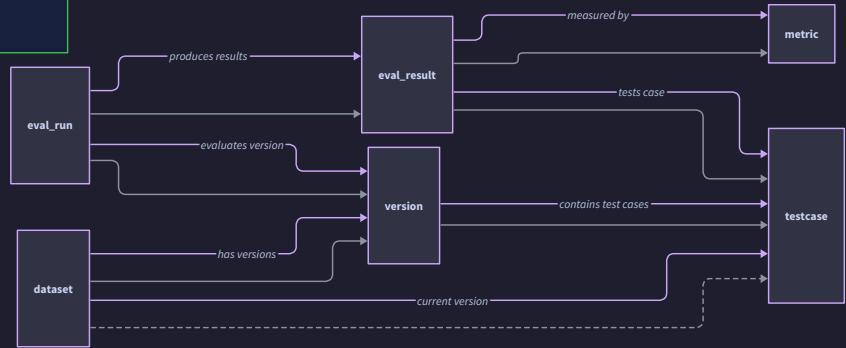
```

Version

```

- "id" UUID
- "dataset_id" UUID
- "version_number" string
- "change_summary" string
- "test_case_count" int
- "created_at" DateTime
- "is_active" bool
+ "get_changes()" Dict
+ "activate()" void

```



Mental Model: The Evaluation Data Ecosystem

Think of the evaluation framework's data model like a scientific research laboratory's record-keeping system. In a lab, researchers maintain detailed records of experiments, specimens, procedures, and results. Each experiment has a specific protocol (like our test cases), produces measurable outcomes (like our evaluation results), and gets catalogued in a versioned system where changes are tracked over time (like our dataset versioning). Just as lab notebooks must be precise, reproducible, and auditable, our data model ensures that every evaluation can be exactly reproduced and compared across time.

The data model forms the backbone of our LLM evaluation framework, defining how test cases, evaluation results, metrics, and dataset versions are structured, stored, and related. This foundation enables reproducible evaluations, reliable comparisons, and comprehensive analysis across different models and time periods.

Test Case Schema

The `TestCase` represents the atomic unit of evaluation - a single prompt-response pair with associated metadata that defines one specific test of model behavior. Think of each test case as a carefully crafted scientific hypothesis: we present a specific input (the prompt) and define what constitutes a correct or acceptable response (the expected output), along with contextual information that helps us understand when and how to apply this test.

Core Design Philosophy

The test case schema balances simplicity with extensibility. Every test case must have the essential components needed for evaluation (prompt and expected output), while the metadata system provides unlimited flexibility for adding domain-specific information without changing the core schema. This design ensures that the framework can handle diverse

evaluation scenarios - from simple question-answering tasks to complex multi-turn conversations - without requiring schema migrations as requirements evolve.

Test Case Structure

Field	Type	Description	Example
case_id	str	Unique identifier for the test case, used for caching and tracking	"tc_001_basic_math"
prompt	str	The input text sent to the language model for evaluation	"What is 2 + 2?"
expected_output	str	The reference answer or ideal response for comparison	"4"
tags	List[str]	Categorical labels for organizing and filtering test cases	["math", "arithmetic", "basic"]
difficulty	str	Complexity level using standardized scale	"easy" "medium" "hard"
metadata	Dict[str, Any]	Flexible key-value storage for domain-specific information	{"topic": "arithmetic", "grade_level": 2}

Design Insight: The `case_id` serves dual purposes - it provides stable references for caching (avoiding re-evaluation when test cases haven't changed) and enables precise tracking of results across dataset versions. Without stable IDs, changing the order of test cases would invalidate cached results.

Metadata Extensibility

The metadata dictionary enables domain-specific extensions without schema changes. Common metadata patterns include:

Metadata Key	Type	Purpose	Example Value
topic	str	Subject matter classification	"customer_support"
complexity	int	Numeric difficulty score (1-10)	7
source	str	Origin of the test case	"human_written"
rubric	Dict[str, str]	Detailed scoring criteria	{"accuracy": "Must be mathematically correct", "format": "Single number only"}
context_length	int	Token count of the prompt	150
expected_tokens	int	Expected length of response	5

Architecture Decision: Flexible Metadata vs. Typed Fields

- **Context:** Need to support diverse evaluation domains without constant schema changes
- **Options Considered:**
 1. Fixed schema with predefined fields for all possible use cases
 2. Completely unstructured data with no required fields
 3. Core required fields plus flexible metadata dictionary
- **Decision:** Core required fields plus flexible metadata dictionary
- **Rationale:** Ensures every test case has the minimum viable information for evaluation while allowing unlimited domain-specific extensions. Fixed schemas become obsolete quickly in ML contexts, while completely unstructured data makes tooling and validation impossible.
- **Consequences:** Enables rapid iteration and domain adaptation, but requires careful metadata documentation to avoid inconsistencies across teams.

Validation Rules

Test case validation ensures data quality and consistency across the evaluation pipeline:

1. **Required Field Validation:** `case_id`, `prompt`, `expected_output`, and `tags` must be non-empty
2. **ID Uniqueness:** `case_id` must be unique within a dataset to prevent cache conflicts
3. **Difficulty Standardization:** `difficulty` must be one of `["easy", "medium", "hard"]` for consistent filtering
4. **Tag Normalization:** Tags are converted to lowercase and whitespace-trimmed to prevent duplicates
5. **Metadata Type Checking:** Metadata values must be JSON-serializable for storage and transmission

Common Test Case Patterns

Different evaluation scenarios require specific test case structures:

Pattern	Description	Prompt Format	Expected Output Format
Question-Answer	Factual questions with single correct answers	"what is the capital of France?"	"Paris"
Classification	Category assignment tasks	"Classify sentiment: I love this product!"	"positive"
Generation	Creative or open-ended outputs	"Write a haiku about coding"	"Bugs in my code flow\nDebugging through the long night\nFinally it works!"
Reasoning	Multi-step logical problems	"If A > B and B > C, what can we conclude?"	"A > C"
Conversation	Multi-turn dialogue simulation	JSON with conversation history	Expected next response

Evaluation Results Model

The evaluation results model captures the complete outcome of running a single test case through the evaluation pipeline. Think of it as the laboratory report that documents not just what happened during an experiment, but also the conditions under which it occurred, the measurements taken, and any observations that might be relevant for future analysis.

Results Architecture

Evaluation results follow a hierarchical structure that mirrors the evaluation process:

Entity	Purpose	Scope	Lifetime
EvaluationRun	Groups all results from a single evaluation session	Entire evaluation batch	Permanent (for historical comparison)
EvaluationResult	Outcome of one test case within a run	Single test case	Permanent
MetricResult	Score from one metric applied to one test case	Single metric calculation	Permanent
ModelResponse	Raw output from the language model	Single LLM call	Cached for performance

EvaluationRun Schema

Field	Type	Description	Example
run_id	str	Unique identifier for this evaluation session	"run_2024_01_15_14_30_gpt4"
dataset_name	str	Name of the dataset being evaluated	"customer_support_v2"
dataset_version	str	Specific version hash of the dataset	"sha256:abc123..."
model_config	LLMProviderConfig	Complete model configuration used	See LLMProviderConfig schema
metrics_config	List[str]	List of metrics applied during evaluation	["exact_match", "semantic_similarity"]
started_at	datetime	Evaluation start timestamp	2024-01-15T14:30:00Z
completed_at	Optional[datetime]	Evaluation completion timestamp (None if failed)	2024-01-15T15:45:30Z
status	str	Current status of the evaluation run	"completed" "running" "failed"
results	List[EvaluationResult]	All individual test case results	Array of EvaluationResult objects
summary_stats	Dict[str, float]	Aggregated metrics across all test cases	{"mean_score": 0.85, "std_dev": 0.12}

EvaluationResult Schema

Field	Type	Description	Example
case_id	str	Reference to the test case that was evaluated	"tc_001_basic_math"
model_response	str	Raw text output from the language model	"The answer is 4."
response_metadata	Dict[str, Any]	Model response details (tokens, latency, etc.)	{"tokens": 6, "latency_ms": 245}
metric_results	Dict[str, MetricResult]	Scores from each applied metric	{"exact_match": MetricResult(...)}
overall_score	float	Weighted average of all metric scores	0.92
evaluation_time	datetime	When this specific test case was evaluated	2024-01-15T14:35:22Z
error_info	Optional[str]	Error message if evaluation failed	None or "API timeout after 30s"

MetricResult Schema

The `MetricResult` encapsulates the outcome of applying a single metric to a model response, providing not just a numeric score but also explanation and debugging information.

Field	Type	Description	Example
score	float	Numeric score between 0.0 and 1.0	0.85
metadata	Dict[str, Any]	Metric-specific calculation details	{"similarity_method": "cosine", "embedding_model": "ada-002"}
explanation	Optional[str]	Human-readable description of the score	"High semantic similarity despite different wording"

Architecture Decision: Detailed Result Storage vs. Aggregated Summaries

- **Context:** Need to balance storage efficiency with debugging capability and future analysis needs
- **Options Considered:**
 1. Store only final aggregated scores to minimize storage
 2. Store complete evaluation traces including intermediate calculations
 3. Store structured results with configurable detail levels
- **Decision:** Store complete structured results with all intermediate data
- **Rationale:** Storage costs are minimal compared to re-evaluation costs, and detailed results enable debugging, analysis, and future metric development. Aggregated-only storage makes it impossible to understand why evaluations failed or to apply new metrics to historical runs.
- **Consequences:** Higher storage requirements, but enables powerful debugging, historical analysis, and metric experimentation without re-running expensive evaluations.

Response Metadata Patterns

Model response metadata captures operational information essential for performance analysis and debugging:

Metadata Key	Type	Purpose	Example
token_count	int	Number of tokens in the response	127
latency_ms	int	Response time in milliseconds	1250
api_version	str	Version of the model API used	"gpt-4-turbo-2024-01-01"
temperature	float	Sampling temperature used	0.7
finish_reason	str	Why the model stopped generating	"stop" "length" "content_filter"
cost_usd	float	Estimated API cost for this call	0.0024

Result Storage Patterns

Evaluation results are stored with multiple access patterns in mind:

1. **By Run ID:** Fast retrieval of complete evaluation sessions for reporting
2. **By Case ID:** Historical tracking of how specific test cases perform over time
3. **By Metric Type:** Analysis of metric-specific trends across evaluations
4. **By Time Range:** Temporal analysis of model performance evolution

Dataset Versioning Model

Dataset versioning provides git-like change tracking for evaluation datasets, enabling reproducible experiments and collaborative dataset development. Think of it as version control for scientific datasets - just as software engineers need to track code changes to understand when bugs were introduced or features were added, ML practitioners need to track dataset changes to understand when evaluation results changed and why.

Versioning Philosophy

The versioning system treats datasets as living documents that evolve through collaborative improvement. Each change is tracked with full provenance information, enabling teams to understand not just what changed, but who made the change, why it was made, and what impact it had on evaluation results. This is crucial for maintaining trust in evaluation results, especially when different team members contribute test cases or when datasets are updated based on model failure analysis.

Version Entity Schema

Field	Type	Description	Example
version_id	str	Unique identifier for this specific version	"v1.2.3" or SHA hash
dataset_name	str	Name of the dataset being versioned	"customer_support_eval"
parent_version	Optional[str]	Previous version this is based on	"v1.2.2"
created_by	str	User or system that created this version	"alice@company.com"
created_at	datetime	When this version was created	2024-01-15T14:30:00Z
commit_message	str	Description of changes in this version	"Added 50 edge cases for numeric reasoning"
change_summary	Dict[str, int]	Statistics about what changed	{"added": 50, "modified": 3, "removed": 1}
test_cases	List[TestCase]	Complete set of test cases in this version	Array of TestCase objects
metadata	Dict[str, Any]	Version-specific information	{"review_status": "approved", "quality_score": 0.95}

Change Tracking

The versioning system tracks changes at the individual test case level, providing detailed diff information:

Change Type	Description	Tracking Method	Impact Analysis
Added	New test cases introduced	Track case_id in added_cases list	Increases dataset coverage
Modified	Existing test cases changed	Store before/after snapshots	May invalidate cached results
Removed	Test cases deleted	Track case_id in removed_cases list	Reduces dataset coverage
Reordered	Test case order changed	Compare position mappings	No impact on results

Version Diff Schema

Field	Type	Description	Example
from_version	str	Starting version for comparison	"v1.2.1"
to_version	str	Ending version for comparison	"v1.2.2"
added_cases	List[str]	Case IDs that were added	["tc_051", "tc_052"]
modified_cases	List[str]	Case IDs that were changed	["tc_010", "tc_023"]
removed_cases	List[str]	Case IDs that were deleted	["tc_007"]
case_changes	Dict[str, Dict]	Detailed field-level changes	See below for structure

Field-Level Change Tracking

For modified test cases, the system tracks exactly which fields changed:

```
{
  "tc_010": {
    "changed_fields": ["expected_output", "tags"],
    "changes": {
      "expected_output": {
        "old": "The answer is 4",
        "new": "4"
      },
      "tags": {
        "old": ["math", "basic"],
        "new": ["math", "arithmetic", "basic"]
      }
    }
  }
}
```

JSON

Architecture Decision: Full Snapshots vs. Delta Storage

- **Context:** Need to balance storage efficiency with query performance for dataset versioning
- **Options Considered:**
 1. Store only deltas between versions (like git objects)
 2. Store complete snapshots of each version
 3. Hybrid approach with snapshots every N versions and deltas in between
- **Decision:** Store complete snapshots with computed deltas for diffs
- **Rationale:** Evaluation datasets are typically small (thousands, not millions of test cases), making storage costs negligible. Complete snapshots enable fast queries and simple backup/restore, while computed deltas provide detailed change analysis. Delta-only storage would require complex reconstruction logic.
- **Consequences:** Higher storage usage, but dramatically simplified queries and reliable data access. Enables instant dataset checkout to any version.

Branching and Merging

The versioning system supports git-like branching for parallel dataset development:

Operation	Purpose	Implementation	Conflict Resolution
Branch	Create parallel development line	Copy current version with new branch name	N/A
Merge	Combine changes from two branches	Three-way merge using common ancestor	Manual resolution required
Cherry-pick	Apply specific test cases from another branch	Copy test cases while preserving history	Automatic if no conflicts
Revert	Undo specific changes	Create new version excluding specified changes	Automatic

Merge Conflict Resolution

When merging branches that modified the same test cases, conflicts must be resolved:

Conflict Type	Detection Method	Resolution Strategy	Example
Same Case Modified	Compare case_id across branches	Present side-by-side diff for manual resolution	Both branches changed expected_output
Case Added/Removed	Check for case_id existence conflicts	Automatic (additions win, removals lose unless explicit)	One branch adds tc_100, other removes it
Tag Conflicts	Compare tag lists for same cases	Union of all tags unless explicitly conflicting	Branch A adds "easy", Branch B adds "difficult"

Dataset Lineage Tracking

The versioning system maintains complete lineage information for audit trails and impact analysis:

Lineage Data	Purpose	Storage Format	Query Patterns
Creation Path	Track how each test case was created	<pre>{"source": "human authored", "author": "alice", "date": "2024-01-15"}</pre>	Find all cases by author
Modification History	Track all changes to each case	List of change records with timestamps	Audit trail for specific cases
Usage Tracking	Record which evaluations used which versions	Evaluation run → dataset version mapping	Impact analysis for version changes
Quality Metrics	Track dataset quality over time	Version → quality score mapping	Quality trend analysis

Common Pitfalls

⚠ Pitfall: Inconsistent Case ID Generation Many implementations generate case IDs based on content hashes or array indices, leading to instability when test cases are reordered or slightly modified. This breaks caching and makes it impossible to track the same logical test case across dataset versions. Instead, use stable, human-meaningful IDs like `"customer_support_billing_001"` that persist even when content changes.

⚠ Pitfall: Unvalidated Metadata Schemas Teams often treat the metadata dictionary as completely unstructured, leading to inconsistent keys, incompatible value types, and broken analysis tools. Establish metadata conventions early and use validation schemas to ensure consistency. For example, always use `snake_case` for keys and document expected value types.

⚠ Pitfall: Missing Evaluation Context Storing results without capturing the exact model configuration, dataset version, and environmental conditions makes it impossible to reproduce evaluations or debug score changes. Always store complete context, including API versions, model parameters, and system configuration.

⚠ Pitfall: Inadequate Change Tracking Simple version numbers without detailed change tracking make it impossible to understand why evaluation scores changed between runs. Implement comprehensive diff tracking that shows exactly which test cases were added, modified, or removed, and how those changes impacted overall scores.

⚠ Pitfall: Ignoring Storage Schema Evolution Hardcoding data structures without considering future extensions leads to painful migrations when new fields are needed. Design schemas with extension points (like metadata dictionaries) and use backward-compatible serialization formats that can handle missing fields gracefully.

Implementation Guidance

The data model implementation requires careful consideration of serialization, validation, and storage patterns to ensure data integrity and performance across the evaluation pipeline.

Technology Recommendations

Component	Simple Option	Advanced Option
Serialization	JSON with Pydantic models	Protocol Buffers with schema evolution
Validation	Pydantic BaseModel with validators	JSON Schema with custom validation rules
Storage	File-based JSON/JSONL	SQLite with indexed queries
Versioning	Directory structure with manifest files	Git-based storage with LFS for large datasets
Caching	In-memory dictionaries	Redis with TTL and compression

File Structure

```

evaluation_framework/
  data/
    models/
      __init__.py           ← Export all data model classes
      test_case.py          ← TestCase class with validation
      evaluation.py         ← EvaluationRun and EvaluationResult classes
      metrics.py            ← MetricResult and related classes
      versioning.py         ← Dataset versioning classes
      serialization.py      ← JSON/dict conversion utilities
    storage/
      __init__.py           ← Dataset persistence layer
      dataset_storage.py    ← Evaluation results persistence
      results_storage.py    ← Version control implementation
  tests/
    test_data_models.py    ← Unit tests for all data model classes
    test_serialization.py  ← Serialization round-trip tests
    test_versioning.py     ← Version control functionality tests

```

Core Data Model Infrastructure

```
# data/models/test_case.py - COMPLETE starter code
```

PYTHON

```
from datetime import datetime

from typing import Dict, List, Any, Optional

from pydantic import BaseModel, Field, validator

import uuid

class TestCase(BaseModel):

    """Represents a single evaluation test case with validation."""

    case_id: str = Field(..., description="Unique identifier for this test case")

    prompt: str = Field(..., min_length=1, description="Input text for the model")

    expected_output: str = Field(..., min_length=1, description="Reference answer")

    tags: List[str] = Field(default_factory=list, description="Categorical labels")

    difficulty: str = Field(default="medium", description="Complexity level")

    metadata: Dict[str, Any] = Field(default_factory=dict, description="Flexible extensions")

    @validator('case_id')

    def validate_case_id(cls, v):

        if not v or v.isspace():

            raise ValueError("case_id cannot be empty or whitespace")

        return v.strip()

    @validator('difficulty')

    def validate_difficulty(cls, v):

        allowed = {"easy", "medium", "hard"}

        if v not in allowed:

            raise ValueError(f"difficulty must be one of {allowed}")

        return v

    @validator('tags')
```

```
def normalize_tags(cls, v):

    return [tag.lower().strip() for tag in v if tag.strip()]


def to_dict(self) -> Dict[str, Any]:

    """Convert to dictionary for serialization."""

    return self.dict()


@classmethod

def from_dict(cls, data: Dict[str, Any]) -> 'TestCase':

    """Create TestCase from dictionary."""

    return cls(**data)


# data/models/evaluation.py - COMPLETE starter code

class MetricResult(BaseModel):

    """Result from applying a single metric to a model response."""

    score: float = Field(..., ge=0.0, le=1.0, description="Score between 0 and 1")

    metadata: Dict[str, Any] = Field(default_factory=dict)

    explanation: Optional[str] = Field(None, description="Human-readable score explanation")


    def to_dict(self) -> Dict[str, Any]:

        return self.dict()


    @classmethod

    def from_dict(cls, data: Dict[str, Any]) -> 'MetricResult':

        return cls(**data)


class EvaluationResult(BaseModel):

    """Result from evaluating a single test case."""


```

```
case_id: str

model_response: str

response_metadata: Dict[str, Any] = Field(default_factory=dict)

metric_results: Dict[str, MetricResult] = Field(default_factory=dict)

overall_score: float = Field(0.0, ge=0.0, le=1.0)

evaluation_time: datetime = Field(default_factory=datetime.utcnow)

error_info: Optional[str] = None

def add_metric_result(self, metric_name: str, result: MetricResult):

    """Add a metric result and update overall score."""

    # TODO: Implement metric result addition

    # TODO: Recalculate overall_score as weighted average of all metrics

    pass

def to_dict(self) -> Dict[str, Any]:

    """Convert to dictionary with datetime serialization."""

    # TODO: Handle datetime serialization properly

    # TODO: Convert MetricResult objects to dictionaries

    pass
```

Dataset Versioning Infrastructure

```
# data/models/versioning.py - Core logic skeleton
```

PYTHON

```
class DatasetVersion(BaseModel):

    """Represents a specific version of an evaluation dataset."""

    version_id: str

    dataset_name: str

    parent_version: Optional[str] = None

    created_by: str

    created_at: datetime = Field(default_factory=datetime.utcnow)

    commit_message: str

    change_summary: Dict[str, int] = Field(default_factory=dict)

    test_cases: List[TestCase] = Field(default_factory=list)

    metadata: Dict[str, Any] = Field(default_factory=dict)

    def add_test_case(self, test_case: TestCase):

        """Add a test case to this version."""

        # TODO 1: Check if case_id already exists in self.test_cases

        # TODO 2: If exists, raise ValueError with clear message

        # TODO 3: Add test_case to self.test_cases list

        # TODO 4: Update change_summary["added"] count

        pass

    def compute_diff(self, other_version: 'DatasetVersion') -> 'VersionDiff':

        """Compute differences between this version and another."""

        # TODO 1: Create sets of case_ids from both versions

        # TODO 2: Compute added_cases = self.case_ids - other.case_ids

        # TODO 3: Compute removed_cases = other.case_ids - self.case_ids

        # TODO 4: Find modified_cases by comparing TestCase content

        # TODO 5: Generate detailed field-level changes for modified cases

        # TODO 6: Return VersionDiff object with all changes
```

```
pass

def to_dict(self) -> Dict[str, Any]:
    """Serialize version with all test cases."""
    # TODO: Handle datetime and nested TestCase serialization
    pass

class VersionDiff(BaseModel):
    """Represents differences between two dataset versions."""

    from_version: str
    to_version: str
    added_cases: List[str] = Field(default_factory=list)
    modified_cases: List[str] = Field(default_factory=list)
    removed_cases: List[str] = Field(default_factory=list)
    case_changes: Dict[str, Dict[str, Any]] = Field(default_factory=dict)

    def has_changes(self) -> bool:
        """Check if there are any changes between versions."""
        # TODO: Return True if any of the change lists are non-empty
        pass

    def impact_score(self) -> float:
        """Calculate impact score based on change magnitude."""
        # TODO 1: Weight different change types (modified > added > removed)
        # TODO 2: Consider percentage of total dataset changed
        # TODO 3: Return score between 0.0 (no changes) and 1.0 (complete replacement)
        pass
```

Serialization Utilities

```
# data/models/serialization.py - COMPLETE helper functions
```

PYTHON

```
import json

from datetime import datetime

from typing import Any, Dict, List, Union

from pathlib import Path


def serialize_datetime(dt: datetime) -> str:
    """Convert datetime to ISO format string."""
    return dt.isoformat() + 'Z'


def deserialize_datetime(dt_str: str) -> datetime:
    """Convert ISO format string back to datetime."""
    if dt_str.endswith('Z'):
        dt_str = dt_str[:-1]
    return datetime.fromisoformat(dt_str)


def save_json(obj: Any, file_path: Union[str, Path]) -> None:
    """Save object as JSON with proper datetime handling."""
    def json_serializer(obj):
        if isinstance(obj, datetime):
            return serialize_datetime(obj)
        if hasattr(obj, 'to_dict'):
            return obj.to_dict()
        raise TypeError(f"Object of type {type(obj)} is not JSON serializable")

    with open(file_path, 'w', encoding='utf-8') as f:
        json.dump(obj, f, default=json_serializer, indent=2, ensure_ascii=False)


def load_json(file_path: Union[str, Path]) -> Any:
    """Load JSON with automatic datetime parsing."""
    with open(file_path, 'r', encoding='utf-8') as f:
        return json.load(f)
```

```
# Validation schema for test case imports

TEST_CASE_SCHEMA = {

    "type": "object",

    "required": ["case_id", "prompt", "expected_output"],

    "properties": {

        "case_id": {"type": "string", "minLength": 1},

        "prompt": {"type": "string", "minLength": 1},

        "expected_output": {"type": "string", "minLength": 1},

        "tags": {"type": "array", "items": {"type": "string"}},

        "difficulty": {"type": "string", "enum": ["easy", "medium", "hard"]},

        "metadata": {"type": "object"}

    }

}
```

Dataset Storage Layer

```
# data/storage/dataset_storage.py - Core logic skeleton

from pathlib import Path

from typing import List, Optional, Dict, Any

import shutil

import hashlib

class DatasetStorage:

    """Handles persistent storage of datasets with versioning."""

    def __init__(self, storage_root: Path):

        self.storage_root = Path(storage_root)

        self.datasets_dir = self.storage_root / "datasets"

        self.versions_dir = self.storage_root / "versions"

        # Create directory structure

        self.datasets_dir.mkdir(parents=True, exist_ok=True)

        self.versions_dir.mkdir(parents=True, exist_ok=True)

    def save_dataset_version(self, version: DatasetVersion) -> str:

        """Save a dataset version and return its hash."""

        # TODO 1: Generate content hash from version.test_cases

        # TODO 2: Create version directory: versions/{dataset_name}/{version_id}/

        # TODO 3: Save test_cases.jsonl with one TestCase per line

        # TODO 4: Save version_metadata.json with version info

        # TODO 5: Update dataset index with new version

        # TODO 6: Return the content hash for caching

        pass

    def load_dataset_version(self, dataset_name: str, version_id: str) -> DatasetVersion:

        """Load a specific dataset version."""


```

```

# TODO 1: Check if version directory exists

# TODO 2: Load version_metadata.json

# TODO 3: Load test_cases.jsonl line by line

# TODO 4: Reconstruct DatasetVersion object

# TODO 5: Validate loaded data integrity

pass

def list_versions(self, dataset_name: str) -> List[str]:
    """List all versions for a dataset, sorted by creation time."""

    # TODO 1: Scan versions/{dataset_name}/ directory

    # TODO 2: Load creation timestamps from metadata

    # TODO 3: Sort by created_at descending (newest first)

    # TODO 4: Return list of version_ids

    pass

def compute_content_hash(self, test_cases: List[TestCase]) -> str:
    """Generate content-based hash for version identification."""

    # TODO 1: Sort test_cases by case_id for consistent hashing

    # TODO 2: Serialize each TestCase to canonical JSON

    # TODO 3: Concatenate all JSON strings

    # TODO 4: Return SHA-256 hash of concatenated content

    pass

```

Validation and Testing Utilities

```
# tests/test_data_models.py - Testing skeleton
```

PYTHON

```
import pytest

from datetime import datetime

from data.models.test_case import TestCase

from data.models.evaluation import EvaluationResult, MetricResult

class TestTestCase:

    """Test suite for TestCase validation and serialization."""

    def test_valid_test_case_creation(self):

        """Test creating a valid test case."""

        # TODO 1: Create TestCase with all required fields

        # TODO 2: Assert all fields are properly set

        # TODO 3: Test default values for optional fields

        pass

    def test_case_id_validation(self):

        """Test case_id validation rules."""

        # TODO 1: Test empty case_id raises ValueError

        # TODO 2: Test whitespace-only case_id raises ValueError

        # TODO 3: Test case_id trimming works correctly

        pass

    def test_serialization_roundtrip(self):

        """Test TestCase serialization and deserialization."""

        # TODO 1: Create TestCase with complex metadata

        # TODO 2: Convert to dict with to_dict()

        # TODO 3: Recreate from dict with from_dict()

        # TODO 4: Assert original and recreated are equal

        pass
```

```

def test_tag_normalization(self):

    """Test tag cleaning and normalization."""

    # TODO 1: Create TestCase with messy tags (mixed case, whitespace)

    # TODO 2: Assert tags are normalized to lowercase and trimmed

    # TODO 3: Test empty tags are removed

    pass


class TestDatasetVersioning:

    """Test suite for dataset versioning functionality."""

    def test_version_diff_computation(self):

        """Test computing diffs between dataset versions."""

        # TODO 1: Create two DatasetVersion objects with known differences

        # TODO 2: Compute diff using compute_diff()

        # TODO 3: Assert added_cases, modified_cases, removed_cases are correct

        # TODO 4: Verify detailed case_changes are accurate

        pass


def test_merge_conflict_detection(self):

    """Test detection of merge conflicts between branches."""

    # TODO 1: Create versions that modify the same test case differently

    # TODO 2: Attempt to compute merge

    # TODO 3: Assert conflicts are properly detected

    # TODO 4: Test conflict resolution workflow

    pass

```

Language-Specific Implementation Hints

- **Pydantic Models:** Use Pydantic for data validation and serialization. The `BaseModel` class provides automatic validation, JSON serialization, and IDE support.
- **Type Hints:** Use comprehensive type hints for all fields and methods. This enables better IDE support and catches errors early.

- **Datetime Handling:** Always use UTC timestamps and ISO format strings for datetime serialization to avoid timezone issues.
- **File Locking:** Use `fcntl.flock()` on Unix or `msvcrt.locking()` on Windows when writing dataset files to prevent corruption from concurrent access.
- **Path Handling:** Use `pathlib.Path` instead of string manipulation for cross-platform file operations.

Milestone Checkpoint

After implementing the data model:

1. **Run validation tests:** `python -m pytest tests/test_data_models.py -v`
2. **Test serialization:** Create a `TestCase`, serialize to JSON, deserialize, and verify equality
3. **Verify versioning:** Create two dataset versions, compute their diff, and check the change summary
4. **Test storage:** Save a dataset version to disk and reload it, ensuring all data is preserved

Expected behavior:

- All Pydantic validation rules should work correctly (try creating invalid test cases)
- JSON serialization should handle datetimes and nested objects properly
- Version diffs should accurately identify changes at the field level
- Storage operations should be atomic and handle concurrent access gracefully

Signs of problems:

- **Serialization errors:** Check datetime handling and ensure all objects have `to_dict()` methods
- **Validation failures:** Verify Pydantic validators are correctly implemented
- **Storage corruption:** Implement proper file locking and atomic write operations
- **Performance issues:** Use efficient data structures and avoid deep copying large datasets unnecessarily

Dataset Management System

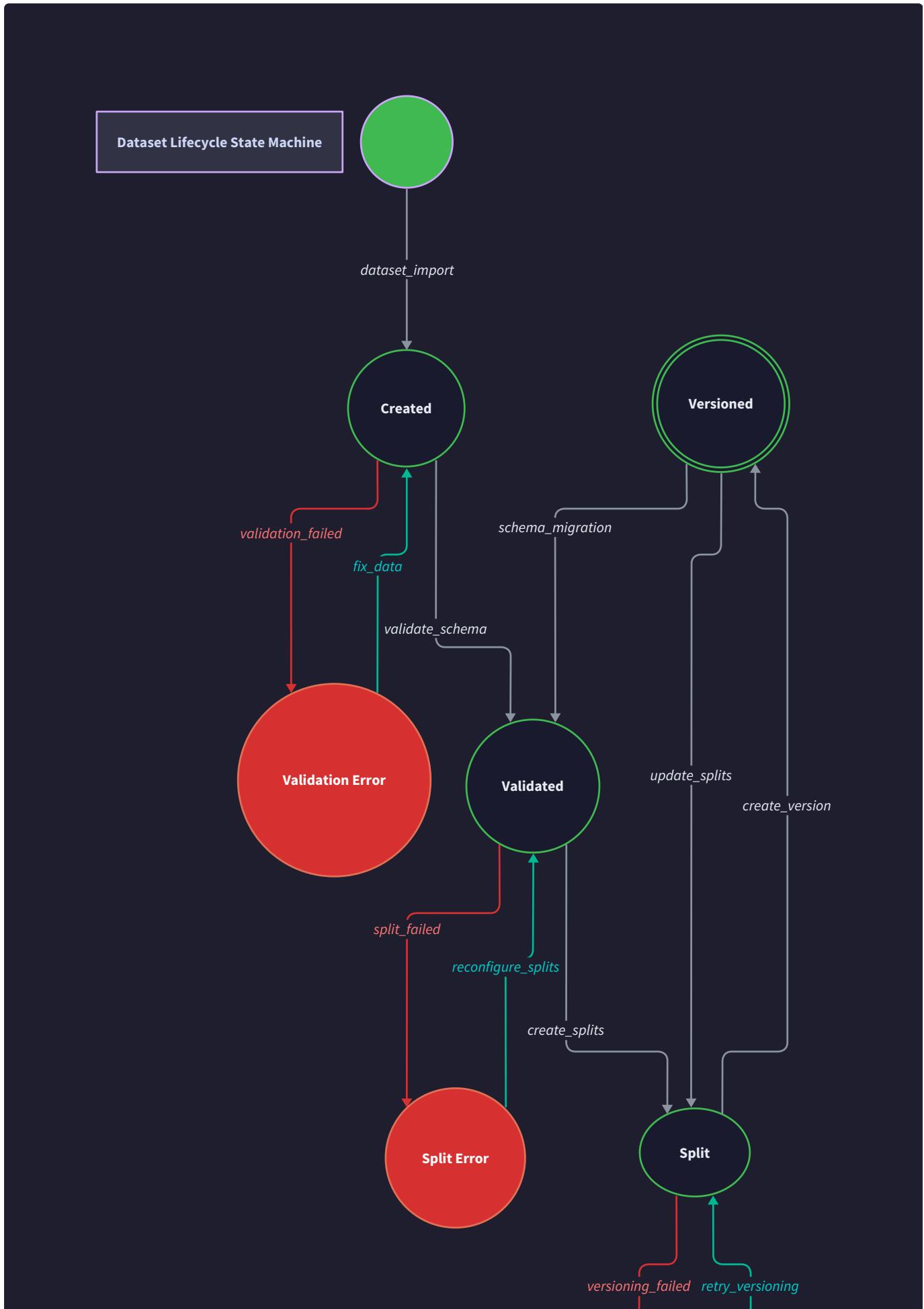
Milestone(s): Milestone 1 (Dataset Management) - This section covers the complete dataset lifecycle from import through versioning and splitting

The dataset management system serves as the foundation of our evaluation framework, much like how a well-organized laboratory manages its test specimens and experimental protocols. Just as a laboratory needs rigorous sample tracking, versioning of experimental procedures, and systematic organization of test conditions, our evaluation framework requires sophisticated dataset management to ensure reproducible, reliable, and scalable LLM evaluations.

Think of the dataset management system as a specialized version control system combined with a data warehouse, specifically designed for evaluation workloads. Unlike traditional databases that optimize for transactional operations, or standard version control systems that handle text files, our system must handle the unique characteristics of evaluation datasets: structured test cases with rich metadata, semantic relationships between similar test cases, and the need for both content-based versioning and temporal tracking of dataset evolution.

The system handles three fundamental responsibilities: ingesting evaluation data from various sources with automatic schema validation, maintaining a complete history of dataset changes with diff capabilities, and organizing datasets into

training and evaluation splits that preserve statistical properties. These capabilities work together to provide the data foundation that enables all other components in the evaluation pipeline.





Dataset Loader

The dataset loader functions as the primary ingestion engine, transforming raw evaluation data from multiple formats into our standardized `TestCase` schema. Think of it as a sophisticated data translator that not only converts between formats but also validates, enriches, and normalizes the incoming data to ensure consistency across all evaluation workflows.

The loader supports three primary input formats: CSV files for tabular data export from spreadsheets and databases, JSON files for structured data with complex nested fields, and JSONL (JSON Lines) format for streaming large datasets efficiently. Each format requires different parsing strategies and validation approaches, but all must produce identical `TestCase` objects that conform to our evaluation schema.

Decision: Multi-Format Support with Automatic Schema Detection

- **Context:** Evaluation teams work with diverse data sources including spreadsheets, databases, annotation tools, and API exports, each producing different file formats
- **Options Considered:**
 1. Support only JSON format and require manual conversion
 2. Support multiple formats with manual schema mapping configuration
 3. Support multiple formats with automatic schema detection and mapping
- **Decision:** Implement automatic schema detection with fallback to manual configuration
- **Rationale:** Automatic detection reduces friction for evaluation teams while maintaining flexibility for edge cases. Manual conversion creates unnecessary barriers, and requiring configuration for every import slows down evaluation iteration cycles
- **Consequences:** Increases loader complexity but dramatically improves user experience. Requires robust error handling for ambiguous schema detection cases

The automatic schema mapping system analyzes input files to identify columns or fields that correspond to our core `TestCase` schema. For CSV files, it uses heuristics based on column names (case-insensitive matching for variations like "prompt", "input", "question") and content analysis (detecting which columns contain long text versus short categorical values). For JSON formats, it performs recursive field analysis to locate nested structures that match our expected schema patterns.

Schema Field	CSV Detection Heuristics	JSON Detection Patterns	Fallback Behavior
<code>prompt</code>	Column names: prompt, input, question, query	Root or nested field matching: prompt, input, text	Use first text column >50 chars avg
<code>expected_output</code>	Column names: expected, answer, output, target	Fields: expected, answer, output, label	Use second text column if available
<code>tags</code>	Column names: tags, categories, labels	Array fields or comma-separated strings	Split on common delimiters (comma, semicolon)
<code>difficulty</code>	Column names: difficulty, level, complexity	String or numeric fields with difficulty keywords	Default to "medium" if not detected
<code>case_id</code>	Column names: id, case_id, test_id	Unique identifier fields	Generate UUID if missing

The validation engine ensures that all loaded test cases conform to the required schema constraints before accepting them into the dataset. This includes checking that essential fields like `prompt` and `expected_output` are non-empty, validating that `tags` contain meaningful categorical information rather than noise, and ensuring that `difficulty` levels map to our standardized taxonomy.

The key insight is that schema validation must happen at ingestion time, not during evaluation execution. Finding malformed test cases during a long-running evaluation wastes computational resources and breaks the evaluation pipeline at the worst possible moment.

Data Structure: TestCase Schema

Field Name	Type	Description	Validation Rules
<code>case_id</code>	str	Unique identifier for the test case	Must be unique within dataset, auto-generated if missing
<code>prompt</code>	str	Input text sent to the language model	Non-empty, length between 1-10000 characters
<code>expected_output</code>	str	Reference answer or expected response	Non-empty, used for scoring and comparison
<code>tags</code>	List[str]	Categorical metadata for grouping and analysis	At least one tag, each tag 1-50 characters
<code>difficulty</code>	str	Complexity level: "easy", "medium", "hard", "expert"	Must match predefined difficulty taxonomy
<code>metadata</code>	Dict[str, Any]	Additional structured data for specialized metrics	JSON-serializable values only

The loader implements a streaming architecture for handling large datasets that exceed available memory. Rather than loading entire files into memory, it processes them in configurable chunks (default 1000 test cases), validates each chunk,

and incrementally builds the complete dataset. This enables processing of datasets with millions of test cases on standard development machines.

Loader Interface Methods

Method Name	Parameters	Returns	Description
<code>load_csv</code>	<code>file_path: Path, schema_mapping: Optional[Dict]</code>	Dataset	Load CSV file with optional manual schema mapping
<code>load_json</code>	<code>file_path: Path, test_case_path: Optional[str]</code>	Dataset	Load JSON file with optional JSONPath for test cases array
<code>load_jsonl</code>	<code>file_path: Path, chunk_size: int = 1000</code>	Dataset	Stream JSONL file processing in chunks
<code>auto_detect_schema</code>	<code>file_path: Path</code>	<code>Dict[str, str]</code>	Analyze file and return detected schema mapping
<code>validate_test_cases</code>	<code>test_cases: List[Dict]</code>	<code>List[str]</code>	Return validation errors for test case list

Version Control System

The version control system provides git-like versioning capabilities specifically optimized for evaluation datasets. Think of it as Git for structured data, where instead of tracking line-by-line changes in text files, we track test case additions, modifications, and deletions with semantic understanding of what each change means for evaluation validity.

Unlike traditional version control systems that operate on file contents, our system versions the semantic structure of datasets. When a test case's expected output changes, the system recognizes this as a potentially significant modification that could affect evaluation results, not just a text edit. This semantic awareness enables powerful features like impact analysis and intelligent merging of concurrent dataset modifications.

Decision: Content-Based Versioning with Semantic Diff

- **Context:** Evaluation datasets evolve continuously as teams discover edge cases, fix incorrect expected outputs, and add new test scenarios
- **Options Considered:**
 1. Simple timestamp-based versioning without change tracking
 2. File-based versioning treating datasets as binary blobs
 3. Content-based versioning with test case level diff tracking
- **Decision:** Implement content-based versioning with semantic diff capabilities
- **Rationale:** Semantic diffs enable impact analysis, rollback of specific changes, and intelligent conflict resolution. Simple versioning loses change context, while file-based versioning can't identify which specific test cases changed
- **Consequences:** Requires sophisticated diff algorithms and more complex storage, but enables advanced dataset management workflows

The versioning system maintains a complete history of dataset evolution using a directed acyclic graph structure similar to Git's commit history. Each `DatasetVersion` represents a snapshot of the dataset at a specific point in time, with references to parent versions enabling branch and merge operations.

Data Structure: DatasetVersion Schema

Field Name	Type	Description	Usage
version_id	str	Content-based hash of the dataset state	Unique identifier, enables deduplication
dataset_name	str	Human-readable dataset identifier	Groups related versions together
parent_version	Optional[str]	Reference to immediate parent version	Enables history traversal and branching
created_by	str	User or system that created this version	Audit trail and responsibility tracking
created_at	datetime	Timestamp of version creation	Temporal ordering and history navigation
commit_message	str	Human-readable description of changes	Documents intent behind modifications
change_summary	Dict[str, int]	Count of added, modified, removed test cases	Quick impact assessment
test_cases	List[TestCase]	Complete set of test cases in this version	Full dataset state for this version
metadata	Dict[str, Any]	Version-specific metadata and tags	Extensible attributes for specialized workflows

The content-based versioning system generates version identifiers by computing cryptographic hashes of the complete dataset contents. This approach ensures that identical datasets always receive identical version IDs, enabling automatic deduplication when multiple users independently create the same dataset state. The hash includes not just the test case contents but also their ordering and metadata, ensuring that semantically different datasets receive different version IDs.

Version Control Operations

Method Name	Parameters	Returns	Description
save_dataset_version	version: DatasetVersion	str	Persist version and return content hash
load_dataset_version	dataset_name: str, version_id: str	DatasetVersion	Retrieve specific dataset version
list_versions	dataset_name: str	List[str]	Get all versions sorted by creation time
compute_diff	from_version: str, to_version: str	VersionDiff	Calculate changes between versions
merge_versions	base: str, branch_a: str, branch_b: str	DatasetVersion	Three-way merge of dataset versions
rollback_changes	version_id: str, changes: List[str]	DatasetVersion	Create new version with specific changes reverted

The diff computation system performs semantic comparison of test cases across versions, identifying additions, modifications, and deletions at the individual test case level. Unlike text-based diffs that show character or line changes, our semantic diffs understand the structure of test cases and can provide meaningful change descriptions.

Data Structure: VersionDiff Schema

Field Name	Type	Description	Usage
from_version	str	Source version identifier	Version being compared from
to_version	str	Target version identifier	Version being compared to
added_cases	List[str]	Test case IDs that were added	New test cases in target version
modified_cases	List[str]	Test case IDs that were changed	Existing test cases with modifications
removed_cases	List[str]	Test case IDs that were deleted	Test cases removed from source version
case_changes	Dict[str, Dict[str, Any]]	Detailed field-level changes for each modified case	Specific modifications within test cases

The system tracks changes at multiple granularities: coarse-grained changes (test case added/modified/removed) for quick impact assessment, and fine-grained changes (specific field modifications) for detailed analysis. This dual-level tracking enables both high-level dataset evolution understanding and precise debugging of evaluation differences across versions.

The critical insight is that evaluation datasets require semantic versioning, not just content versioning. A single character change in an expected output can fundamentally alter evaluation results, so the versioning system must understand and track these semantic changes explicitly.

Dataset Splitting

The dataset splitting system provides sophisticated algorithms for dividing datasets into training, testing, and validation subsets while preserving important statistical properties. Think of it as a scientific sampling system that ensures each subset remains representative of the overall population, preventing evaluation bias that could lead to incorrect conclusions about model performance.

Traditional random splitting works adequately for homogeneous datasets, but evaluation datasets often contain structured heterogeneity: different difficulty levels, topic categories, input formats, and edge case patterns. Simple random splitting might accidentally concentrate all difficult cases in the training set or place all examples from specific categories in the validation set, leading to unreliable evaluation results.

Decision: Stratified Sampling with Multi-Attribute Support

- **Context:** Evaluation datasets contain multiple categorical and continuous attributes that affect test case difficulty and model performance patterns
- **Options Considered:**
 1. Simple random splitting ignoring test case attributes
 2. Stratified sampling on a single attribute (e.g., difficulty)
 3. Multi-attribute stratified sampling with proportional allocation
- **Decision:** Implement multi-attribute stratified sampling with configurable stratification keys
- **Rationale:** Multi-attribute stratification ensures representative splits across all important dimensions, preventing evaluation bias. Single-attribute stratification misses interactions between attributes, while random splitting provides

no guarantees about subset representativeness

- **Consequences:** Requires more complex splitting algorithms and may not always achieve exact split ratios, but provides much more reliable evaluation results

The stratified sampling algorithm works by creating strata (groups) based on combinations of categorical attributes, then sampling from each stratum proportionally to maintain the overall distribution. For example, if the original dataset contains 30% easy cases, 50% medium cases, and 20% hard cases, each split will maintain approximately the same proportions.

Stratified Splitting Algorithm

1. **Analyze the dataset** to identify all unique combinations of stratification attributes, creating a stratum for each combination
2. **Calculate target proportions** for each stratum based on their frequency in the complete dataset
3. **Determine target sizes** for each split (train/test/validation) based on the configured split ratios
4. **Allocate test cases** from each stratum to splits proportionally, ensuring minimum representation thresholds are met
5. **Handle remainder cases** when proportional allocation doesn't divide evenly by using randomized assignment weighted by stratum size
6. **Validate split quality** by computing distribution statistics and flagging splits that deviate significantly from target proportions

Dataset Splitting Interface

Method Name	Parameters	Returns	Description
<code>train_test_split</code>	<code>test_size: float, stratify_by: List[str]</code>	<code>Tuple[Dataset, Dataset]</code>	Split into training and testing datasets
<code>train_test_val_split</code>	<code>test_size: float, val_size: float, stratify_by: List[str]</code>	<code>Tuple[Dataset, Dataset, Dataset]</code>	Three-way split with validation set
<code>k_fold_split</code>	<code>k: int, stratify_by: List[str]</code>	<code>List[Tuple[Dataset, Dataset]]</code>	Generate k training/validation pairs for cross-validation
<code>temporal_split</code>	<code>split_date: datetime</code>	<code>Tuple[Dataset, Dataset]</code>	Split based on test case creation timestamps
<code>validate_split_quality</code>	<code>splits: List[Dataset], stratify_by: List[str]</code>	<code>Dict[str, float]</code>	Compute distribution similarity metrics

The system supports both proportional and absolute split specifications. Proportional splits specify percentages (e.g., 70% train, 20% test, 10% validation), while absolute splits specify exact counts (e.g., 1000 test cases, remainder for training).

This flexibility accommodates different evaluation scenarios: research experiments often use proportional splits, while production evaluations may require fixed-size test sets.

Split Quality Validation Metrics

Metric Name	Formula	Interpretation	Threshold
Chi-Square Statistic	$\sum((\text{observed} - \text{expected})^2 / \text{expected})$	Distribution similarity between splits	< 0.05 for good splits
Jensen-Shannon Divergence	Symmetric KL divergence variant	Information-theoretic distribution distance	< 0.1 for similar distributions
Cramér's V	$\sqrt{(\chi^2 / (n \times \min(k-1, r-1)))}$	Effect size of distribution differences	< 0.3 for acceptable splits
Minimum Stratum Size	$\min(\text{count per stratum per split})$	Ensures adequate representation	≥ 5 cases per stratum per split

The temporal splitting capability handles datasets where test cases have creation timestamps, enabling evaluation of model performance over time. This is crucial for detecting concept drift or evaluating how models perform on newer types of questions that weren't available during training dataset creation.

Common Pitfalls in Dataset Splitting

⚠ Pitfall: Ignoring Stratification Attributes Many teams perform simple random splits without considering test case attributes, leading to biased evaluation results. For example, if all "expert" difficulty cases randomly end up in the training set, the test set won't properly evaluate model performance on difficult cases. Always identify the key attributes that affect model performance and stratify on those attributes.

⚠ Pitfall: Insufficient Stratum Representation When stratifying on multiple attributes, some attribute combinations may have very few test cases, leading to strata with only 1-2 examples. These cases can't be split meaningfully across train/test sets. The system should either merge small strata with similar ones or flag datasets that don't have sufficient diversity for the desired split configuration.

⚠ Pitfall: Temporal Leakage in Time-Series Data For datasets containing temporal information, random splitting can create temporal leakage where test cases from earlier time periods are used to evaluate models on later time periods. Always use temporal splitting for time-sensitive evaluation scenarios, placing earlier cases in training and later cases in testing.

⚠ Pitfall: Not Validating Split Quality Teams often assume that split algorithms produce representative subsets without verification. Always compute and review split quality metrics, especially for datasets with complex attribute distributions. Poor splits can invalidate entire evaluation studies.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
File Parsing	Python <code>csv</code> + <code>json</code> modules	<code>pandas</code> with <code>pyarrow</code> backend for performance
Schema Validation	Manual validation with <code>jsonschema</code>	<code>pydantic</code> models with automatic validation
Content Hashing	<code>hashlib.sha256</code> with JSON serialization	<code>xxhash</code> with custom serialization for speed
Data Storage	File-based with <code>pickle</code> serialization	<code>sqlite</code> database with JSON columns
Stratified Sampling	Custom implementation with <code>random</code>	<code>scikit-learn.model_selection.train_test_split</code>

Recommended File Structure

```

evaluation_framework/
  dataset_management/
    __init__.py           ← Public API exports
    loader.py             ← Dataset loading and validation
    version_control.py   ← Versioning and diff computation
    splitting.py          ← Train/test/validation splits
    storage.py            ← Persistence layer
    schemas.py            ← Data model definitions
    exceptions.py         ← Custom exception classes
  tests/
    test_loader.py        ← Unit tests for loader
    test_version_control.py ← Version control tests
    test_splitting.py     ← Split quality tests
  fixtures/
    sample.csv            ← Sample datasets for testing
    sample.json
    sample.jsonl

```

Infrastructure Starter Code

```
# schemas.py - Complete data model definitions

from dataclasses import dataclass, field

from datetime import datetime

from pathlib import Path

from typing import Any, Dict, List, Optional

import hashlib

import json

import uuid

@dataclass

class TestCase:

    """Individual test case with all required evaluation metadata."""

    case_id: str

    prompt: str

    expected_output: str

    tags: List[str]

    difficulty: str

    metadata: Dict[str, Any] = field(default_factory=dict)

    def __post_init__(self):

        """Validate test case after creation."""

        if not self.case_id:

            self.case_id = str(uuid.uuid4())

        if not self.prompt.strip():

            raise ValueError("Test case prompt cannot be empty")

        if not self.expected_output.strip():

            raise ValueError("Test case expected_output cannot be empty")

        if not self.tags:

            raise ValueError("Test case must have at least one tag")

        if self.difficulty not in ["easy", "medium", "hard", "expert"]:
```

```
        raise ValueError(f"Invalid difficulty: {self.difficulty}")

def to_dict(self) -> Dict[str, Any]:
    """Serialize to dictionary for storage and hashing."""
    return {
        "case_id": self.case_id,
        "prompt": self.prompt,
        "expected_output": self.expected_output,
        "tags": self.tags,
        "difficulty": self.difficulty,
        "metadata": self.metadata
    }

@classmethod
def from_dict(cls, data: Dict[str, Any]) -> 'TestCase':
    """Deserialize from dictionary."""
    return cls(**data)

@dataclass
class Dataset:
    """Collection of test cases with metadata."""

    name: str
    test_cases: List[TestCase]
    version: str
    description: str
    created_at: datetime = field(default_factory=datetime.now)

    def add_test_case(self, test_case: TestCase):
        """Add a test case to the dataset."""
        # Check for duplicate case_id
```

```
existing_ids = {tc.case_id for tc in self.test_cases}

if test_case.case_id in existing_ids:

    raise ValueError(f"Duplicate case_id: {test_case.case_id}")

self.test_cases.append(test_case)

def to_dict(self) -> Dict[str, Any]:
    """Serialize dataset for storage."""
    return {
        "name": self.name,
        "test_cases": [tc.to_dict() for tc in self.test_cases],
        "version": self.version,
        "description": self.description,
        "created_at": self.created_at.isoformat()
    }

@dataclass
class VersionDiff:
    """Represents changes between two dataset versions."""
    from_version: str
    to_version: str
    added_cases: List[str]
    modified_cases: List[str]
    removed_cases: List[str]
    case_changes: Dict[str, Dict[str, Any]]

    def has_changes(self) -> bool:
        """Check if this diff contains any changes."""
        return bool(self.added_cases or self.modified_cases or self.removed_cases)

    def impact_score(self) -> float:
```

```

    """Calculate impact score based on change magnitude."""

    total_changes = len(self.added_cases) + len(self.modified_cases) + len(self.removed_cases)

    # Weight modifications higher than additions/removals

    weighted_score = len(self.added_cases) + len(self.removed_cases) + (len(self.modified_cases)
* 1.5)

    return weighted_score

def compute_content_hash(test_cases: List[TestCase]) -> str:

    """Generate content-based hash for dataset version identification."""

    # Sort test cases by case_id for consistent hashing

    sorted_cases = sorted(test_cases, key=lambda tc: tc.case_id)

    content_json = json.dumps([tc.to_dict() for tc in sorted_cases], sort_keys=True)

    return hashlib.sha256(content_json.encode('utf-8')).hexdigest()

# Storage utilities

class DatasetStorage:

    """File-based storage for datasets and versions."""

    def __init__(self, storage_path: Path):

        self.storage_path = storage_path

        self.storage_path.mkdir(parents=True, exist_ok=True)

        self.datasets_path = self.storage_path / "datasets"

        self.versions_path = self.storage_path / "versions"

        self.datasets_path.mkdir(exist_ok=True)

        self.versions_path.mkdir(exist_ok=True)

    def save_dataset(self, dataset: Dataset) -> str:

        """Save dataset and return file path."""

        dataset_file = self.datasets_path / f"{dataset.name}_{dataset.version}.json"

        with open(dataset_file, 'w') as f:

            json.dump(dataset.to_dict(), f, indent=2)

```

```
    return str(dataset_file)

def load_dataset(self, name: str, version: str) -> Dataset:
    """Load dataset by name and version."""
    dataset_file = self.datasets_path / f"{name}_{version}.json"
    with open(dataset_file, 'r') as f:
        data = json.load(f)

    # Reconstruct test cases
    test_cases = [TestCase.from_dict(tc_data) for tc_data in data["test_cases"]]
    return Dataset(
        name=data["name"],
        test_cases=test_cases,
        version=data["version"],
        description=data["description"],
        created_at=datetime.fromisoformat(data["created_at"])
    )
```

Core Logic Skeleton Code

```
# loader.py - Dataset loading with schema detection
```

PYTHON

```
import csv

import json

from pathlib import Path

from typing import Dict, List, Optional, Any

import pandas as pd

class DatasetLoader:

    """Handles loading datasets from various file formats with schema detection."""

    def __init__(self):

        self.common_prompt_names = {"prompt", "input", "question", "query", "text"}

        self.common_output_names = {"expected", "answer", "output", "target", "label"}

        self.common_tag_names = {"tags", "categories", "labels", "topics"}

        self.common_difficulty_names = {"difficulty", "level", "complexity", "hardness"}


    def load_csv(self, file_path: Path, schema_mapping: Optional[Dict[str, str]] = None) -> Dataset:

        """Load CSV file with optional manual schema mapping."""

        # TODO 1: Read CSV file using pandas or csv module

        # TODO 2: If schema_mapping not provided, call auto_detect_schema()

        # TODO 3: Apply schema mapping to rename/select columns

        # TODO 4: Convert each row to TestCase using _row_to_test_case()

        # TODO 5: Create and return Dataset object with all test cases

        # Hint: Handle missing values and empty cells appropriately

        pass


    def auto_detect_schema(self, file_path: Path) -> Dict[str, str]:

        """Analyze file and return detected schema mapping."""

        # TODO 1: Read first 100 rows to sample the data structure

        # TODO 2: For each column, compute heuristic scores for each field type
```

```

# TODO 3: Assign column with highest score to each required field

# TODO 4: Validate that all required fields have been mapped

# TODO 5: Return mapping dict like {"prompt": "Question", "expected_output": "Answer"}
```

Hint: Use string similarity and content analysis for scoring

```
pass
```



```
def _detect_field_type(self, column_name: str, sample_values: List[str]) -> Dict[str, float]:
    """Score how likely a column is to be each field type."""

    # TODO 1: Score column name similarity to common field names

    # TODO 2: Analyze sample values (length, patterns, content type)

    # TODO 3: Return scores dict like {"prompt": 0.8, "expected_output": 0.1, ...}

    # Hint: Long text columns likely prompts/outputs, short text likely tags/difficulty

    pass
```



```
def load_jsonl(self, file_path: Path, chunk_size: int = 1000) -> Dataset:
    """Stream JSONL file processing in chunks."""

    # TODO 1: Open file and create empty test_cases list

    # TODO 2: Read file line by line, parsing each JSON object

    # TODO 3: Convert each JSON object to TestCase

    # TODO 4: Process in chunks to avoid memory issues with large files

    # TODO 5: Validate test cases periodically and handle errors gracefully

    # Hint: Use yield or generators for memory efficiency

    pass
```



```
# version_control.py - Git-like versioning for datasets
```

```
class DatasetVersionControl:
    """Provides git-like versioning capabilities for evaluation datasets."""

    def __init__(self, storage: DatasetStorage):
        self.storage = storage
```

```
        self._version_graph = {} # version_id -> DatasetVersion


    def save_dataset_version(self, version: DatasetVersion) -> str:
        """Save dataset version and return content hash."""

        # TODO 1: Compute content hash from test cases using compute_content_hash()

        # TODO 2: Set version.version_id to the computed hash

        # TODO 3: Save version metadata to storage

        # TODO 4: Update internal version graph for traversal

        # TODO 5: Return the version_id (content hash)

        # Hint: Check if version already exists to avoid duplicates

        pass


    def compute_diff(self, from_version: str, to_version: str) -> VersionDiff:
        """Compute differences between dataset versions."""

        # TODO 1: Load both dataset versions from storage

        # TODO 2: Create case_id -> TestCase mappings for both versions

        # TODO 3: Find added cases (in to_version but not from_version)

        # TODO 4: Find removed cases (in from_version but not to_version)

        # TODO 5: Find modified cases (same case_id but different content)

        # TODO 6: For modified cases, compute field-level changes

        # TODO 7: Return VersionDiff object with all changes

        # Hint: Use set operations for efficient difference computation

        pass


# splitting.py - Stratified dataset splitting

from sklearn.model_selection import train_test_split

from collections import defaultdict, Counter

import random

from typing import Tuple
```

```
class DatasetSplitter:

    """Handles train/test/validation splits with stratification support."""

    def __init__(self, random_state: int = 42):
        self.random_state = random_state
        random.seed(random_state)

    def train_test_split(self, dataset: Dataset, test_size: float,
                        stratify_by: List[str]) -> Tuple[Dataset, Dataset]:
        """Split dataset into training and testing sets."""

        # TODO 1: Extract stratification keys from each test case
        # TODO 2: Create strata by grouping test cases with same key combinations
        # TODO 3: For each stratum, split proportionally between train and test
        # TODO 4: Ensure minimum representation (at least 1 case per stratum per split)
        # TODO 5: Create new Dataset objects for train and test splits
        # TODO 6: Validate split quality using _validate_split_quality()

        # Hint: Use Counter to track stratum frequencies
        pass

    def _create_stratification_key(self, test_case: TestCase, stratify_by: List[str]) -> str:
        """Create stratification key from test case attributes."""

        # TODO 1: Extract values for each attribute in stratify_by list
        # TODO 2: Handle missing attributes gracefully with default values
        # TODO 3: Combine into a single string key for grouping
        # TODO 4: Ensure consistent key format across all test cases

        # Hint: Use tuple conversion to string for consistent ordering
        pass

    def _validate_split_quality(self, original: Dataset, splits: List[Dataset],
```

```
        stratify_by: List[str]) -> Dict[str, float]:\n\n        """Compute distribution similarity metrics between original and splits."""\n\n        # TODO 1: Compute attribute distributions for original dataset\n\n        # TODO 2: Compute attribute distributions for each split\n\n        # TODO 3: Calculate chi-square statistics for distribution similarity\n\n        # TODO 4: Compute Jensen-Shannon divergence between distributions\n\n        # TODO 5: Return metrics dict with quality scores\n\n        # Hint: Use scipy.stats for statistical tests\n\n        pass
```

Milestone Checkpoint

After implementing the dataset management system, verify functionality with these tests:

1. Dataset Loading Test:

```
python -m pytest tests/test_loader.py -v
```

BASH

Expected: All format parsers (CSV, JSON, JSONL) successfully load sample datasets and auto-detect schemas correctly.

2. Version Control Test:

```
python -c "
from dataset_management import DatasetVersionControl, Dataset, TestCase

# Create test dataset

dataset = Dataset('test', [TestCase('1', 'prompt', 'output', ['tag'], 'easy')], 'v1', 'test')

# Test versioning

vc = DatasetVersionControl()

version_id = vc.save_dataset_version(dataset)

print(f'Saved version: {version_id}')


# Test diff computation

diff = vc.compute_diff(version_id, version_id)

print(f'Self-diff has changes: {diff.has_changes()}') # Should be False

"
"
```

3. Stratified Splitting Test:

```

python -c "
from dataset_management import DatasetSplitter, Dataset, TestCase

# Create test dataset with different difficulties

cases = [TestCase(f'{i}', f'prompt{i}', f'output{i}', ['tag'],
                  'easy' if i < 5 else 'hard') for i in range(10)]

dataset = Dataset('test', cases, 'v1', 'test')

# Test stratified split

splitter = DatasetSplitter()

train, test = splitter.train_test_split(dataset, test_size=0.3, stratify_by=['difficulty'])

print(f'Train size: {len(train.test_cases)}, Test size: {len(test.test_cases)})')

# Both splits should have easy and hard cases proportionally

"

```

Debugging Tips

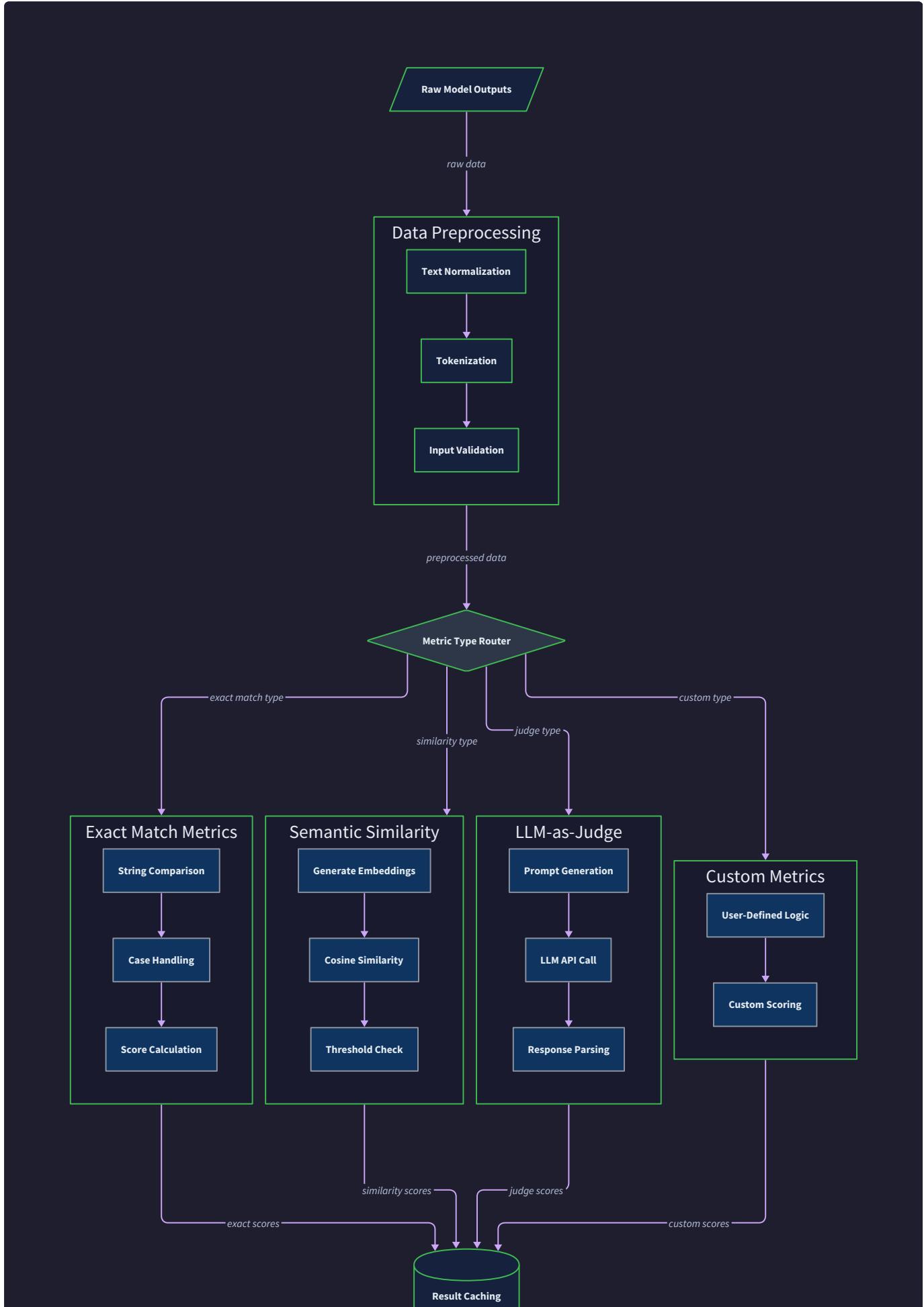
Symptom	Likely Cause	How to Diagnose	Fix
Schema auto-detection fails	Column names don't match heuristics	Print detected column scores and mappings	Add more column name variations to detection lists
Version diff shows all cases as modified	Inconsistent test case ordering or serialization	Check content hash generation	Sort test cases by case_id before hashing
Stratified split fails with small strata	Not enough cases for each attribute combination	Count cases per stratum before splitting	Merge similar strata or increase minimum stratum size
Large datasets cause memory errors	Loading entire dataset into memory	Monitor memory usage during loading	Implement streaming/chunked processing

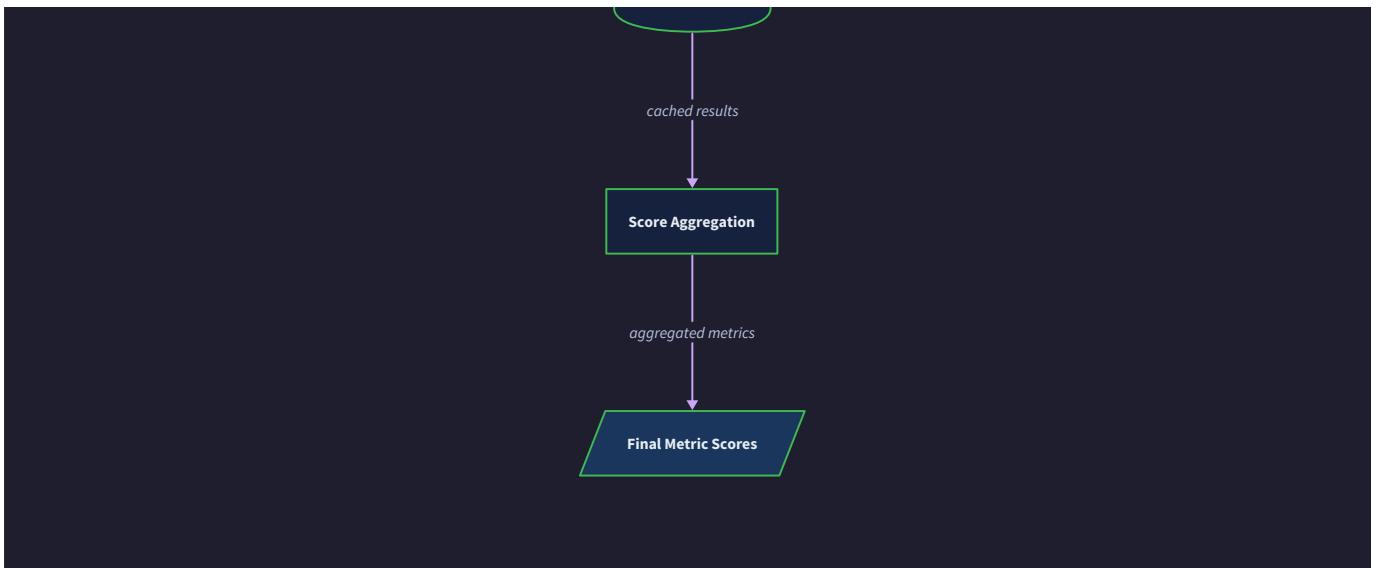
Metrics Engine

Milestone(s): Milestone 2 (Evaluation Metrics) - This section covers the complete metrics computation system including exact match, semantic similarity, LLM-as-judge, and custom metrics

The metrics engine is the heart of our evaluation framework - it transforms raw model outputs into meaningful scores that guide AI system improvement. Think of the metrics engine as a sophisticated grading system, similar to how a university might evaluate student essays using multiple criteria: some assignments get scored by exact answer matching (like math problems), others by semantic understanding (like literature analysis), and still others by expert human judgment (like creative writing). Our engine mirrors this flexibility by supporting multiple evaluation approaches within a unified, extensible architecture.

The core architectural challenge lies in creating a system that can seamlessly handle vastly different metric types - from simple string comparison to complex neural embedding calculations to LLM-powered evaluation - while maintaining consistent interfaces, reliable caching, and meaningful score aggregation. Each metric type has unique computational requirements, error modes, and performance characteristics, yet they must integrate smoothly into the same evaluation pipeline.





Metric Interface Design

The foundation of our extensible metrics system rests on a common interface that abstracts the scoring process while accommodating the diverse computational patterns of different metric types. Think of this interface as a universal translator that allows metrics as different as exact string matching and neural similarity computation to speak the same language within our evaluation framework.

BaseMetric Abstract Class

Every metric in our system inherits from `BaseMetric`, which defines the essential contract for score computation. The interface design balances simplicity for implementers with sufficient flexibility to handle the computational complexity of modern evaluation approaches.

Method Name	Parameters	Returns	Description
<code>compute</code>	<code>model_output: str</code> , <code>expected_output: str</code> , <code>test_case: TestCase</code>	<code>MetricResult</code>	Core scoring logic that compares model output against expected result
<code>is_applicable</code>	<code>test_case: TestCase</code>	<code>bool</code>	Determines if this metric should run for the given test case based on tags/metadata
<code>get_name</code>	<code>None</code>	<code>str</code>	Returns unique identifier for this metric type
<code>get_description</code>	<code>None</code>	<code>str</code>	Human-readable description of what this metric measures
<code>requires_preprocessing</code>	<code>None</code>	<code>bool</code>	Whether metric needs text normalization before scoring
<code>get_preprocessing_steps</code>	<code>None</code>	<code>List[str]</code>	List of preprocessing operations (e.g., "normalize_whitespace", "lowercase")

The `compute` method serves as the primary entry point, receiving the model's actual output, the expected reference output, and the complete test case context. This design enables metrics to access rich metadata like difficulty level, content tags, and custom annotations that might influence scoring behavior.

Decision: Three-Parameter Compute Interface

- **Context:** Metrics need different amounts of context - some only need output strings, others need test case metadata
- **Options Considered:**
 1. Simple two-parameter interface (output, expected)
 2. Three-parameter interface (output, expected, test_case)
 3. Context object with all evaluation data
- **Decision:** Three-parameter interface with explicit `test_case` parameter
- **Rationale:** Provides necessary context without excessive coupling, makes dependencies explicit
- **Consequences:** Slightly more complex interface but enables context-aware metrics and future extensibility

MetricResult Data Structure

The standardized return type encapsulates all information from a single metric evaluation, designed to support both simple numeric scores and rich explanatory metadata that aids in debugging and analysis.

Field Name	Type	Description
<code>score</code>	<code>float</code>	Primary numeric score, normalized to 0.0-1.0 range for consistency
<code>metadata</code>	<code>Dict[str, Any]</code>	Additional metric-specific data (confidence, intermediate values, etc.)
<code>explanation</code>	<code>Optional[str]</code>	Human-readable explanation of how the score was derived
<code>computation_time</code>	<code>float</code>	Milliseconds spent computing this metric for performance monitoring
<code>error_info</code>	<code>Optional[str]</code>	Error details if computation failed (score set to 0.0 on failure)
<code>preprocessing_applied</code>	<code>List[str]</code>	Record of text transformations applied before scoring

The score normalization to 0.0-1.0 range enables meaningful aggregation across different metric types. A semantic similarity metric computing cosine distances and an exact match metric returning binary values can both contribute equally to overall evaluation scores.

Metric Applicability System

The `is_applicable` method implements a sophisticated filtering system that determines which metrics should evaluate which test cases. This prevents inappropriate metric application (like running code-execution metrics on creative writing prompts) and enables targeted evaluation strategies.

Applicability Pattern	Implementation Strategy	Example Use Case
Tag-based filtering	Check for required tags in <code>test_case.tags</code>	Semantic similarity only for "reasoning" tasks
Difficulty-based filtering	Compare against <code>test_case.difficulty</code> level	Complex metrics skip "easy" baseline cases
Content-type filtering	Inspect expected output format/structure	Code metrics require valid syntax in expected output
Metadata-driven filtering	Custom logic based on <code>test_case.metadata</code>	LLM-as-judge only when reference answer exists

The applicability system serves as quality gates, ensuring metrics only run where they provide meaningful signal rather than noise.

Preprocessing Pipeline Integration

Many metrics benefit from text normalization before scoring - removing extra whitespace, standardizing case, or stripping formatting artifacts. The interface design makes preprocessing explicit and trackable rather than hidden within metric implementations.

Standard preprocessing operations include:

1. **Whitespace normalization:** Convert multiple spaces/tabs/newlines to single spaces
2. **Case normalization:** Convert to lowercase for case-insensitive comparison
3. **Punctuation handling:** Remove or normalize punctuation marks
4. **Unicode normalization:** Handle different Unicode representations of same characters
5. **Format stripping:** Remove markdown, HTML, or other formatting artifacts

Each applied preprocessing step gets recorded in the `MetricResult.preprocessing_applied` field, enabling debugging when preprocessing changes affect scores unexpectedly.

Built-in Metrics

Our framework provides a comprehensive suite of built-in metrics that handle the most common evaluation scenarios. Each metric represents years of research in computational linguistics and has been battle-tested across diverse AI applications.

Exact Match Scorer

The exact match metric serves as the foundation for deterministic evaluation scenarios where precision matters more than flexibility. Think of it as a spelling test - either the answer is exactly correct, or it isn't.

The implementation supports multiple matching modes to handle common real-world variations:

Matching Mode	Preprocessing Applied	Use Case
strict	None	Code generation, formal mathematical expressions
normalized	Whitespace + case normalization	Natural language with formatting variations
fuzzy	Full preprocessing pipeline	Human-written reference answers with inconsistent formatting

```
class ExactMatchMetric(BaseMetric):
    def compute(self, model_output: str, expected_output: str, test_case: TestCase) -> MetricResult:
        # Implementation handles three matching modes based on test case configuration
```

PYTHON

The exact match metric also implements **substring matching** for scenarios where the correct answer appears within a longer model response. This proves essential for evaluating models that tend to provide explanatory context alongside their core answers.

BLEU Score Calculator

BLEU (Bilingual Evaluation Understudy) measures n-gram overlap between model output and reference text, originally developed for machine translation but widely applicable to text generation tasks. Think of BLEU as measuring how many "phrases" the model got right, where phrases can be single words (1-grams), word pairs (2-grams), or longer sequences.

N-gram Level	Weight	Contribution	Example
1-gram (unigrams)	0.25	Word-level overlap	"the", "cat", "sat"
2-gram (bigrams)	0.25	Phrase-level accuracy	"the cat", "cat sat"
3-gram (trigrams)	0.25	Short phrase fluency	"the cat sat"
4-gram (4-grams)	0.25	Sentence-level structure	"the cat sat on"

The BLEU calculation incorporates a **brevity penalty** that prevents gaming the metric by generating very short outputs that achieve high precision but low recall. The penalty function exponentially reduces scores for outputs significantly shorter than the reference text.

Our implementation handles multiple reference answers by computing BLEU against each reference and taking the maximum score, reflecting the common scenario where multiple valid answers exist for the same prompt.

ROUGE Score Implementation

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) focuses on recall rather than precision, measuring what fraction of the reference content appears in the model output. This proves especially valuable for summarization and information extraction tasks where completeness matters more than brevity.

ROUGE Variant	Measurement Focus	Calculation Method	Best Use Case
ROUGE-N	N-gram recall	Count of matching n-grams / total n-grams in reference	Content coverage assessment
ROUGE-L	Longest common subsequence	LCS length / reference length	Structural similarity evaluation
ROUGE-W	Weighted longest common subsequence	Weighted LCS with gap penalties	Order-sensitive content evaluation
ROUGE-S	Skip-gram matching	Co-occurrence patterns allowing gaps	Flexible content matching

The ROUGE-L variant proves particularly valuable for evaluating structured outputs like lists or step-by-step procedures, where the order and completeness of information matter more than exact wording.

Semantic Similarity Scorer

Semantic similarity metrics move beyond surface-level text comparison to evaluate meaning preservation. Think of this as the difference between a human teacher who understands that "large" and "big" convey the same concept versus a simple string matcher that sees them as completely different.

Our implementation supports multiple embedding approaches:

Embedding Model	Dimension	Strengths	Computational Cost
Sentence-BERT	384-768	General purpose, fast inference	Low
OpenAI text-embedding-3-small	1536	Strong semantic understanding	Medium
OpenAI text-embedding-3-large	3072	Highest quality semantic representation	High
Domain-specific fine-tuned	Variable	Specialized vocabulary understanding	Variable

The scoring process follows a multi-step pipeline:

- Text preprocessing:** Remove formatting artifacts that could skew embeddings
- Embedding generation:** Convert both texts to dense vector representations
- Similarity computation:** Calculate cosine similarity between embedding vectors
- Score normalization:** Map similarity values to 0.0-1.0 range with appropriate thresholds

Critical Insight: Raw cosine similarity scores require calibration - a score of 0.8 might indicate strong similarity in one domain but weak similarity in another. Our implementation includes domain-specific threshold tuning.

Calibration and Threshold Management

Each semantic similarity model requires careful calibration to produce meaningful scores. We implement an automated calibration system that analyzes score distributions across golden examples to establish domain-specific thresholds.

Similarity Range	Semantic Interpretation	Score Mapping	Confidence Level
0.95-1.0	Nearly identical meaning	1.0	Very High
0.85-0.94	Strong semantic overlap	0.8-1.0	High
0.70-0.84	Moderate similarity	0.6-0.8	Medium
0.55-0.69	Weak similarity	0.3-0.6	Low
0.0-0.54	Little to no similarity	0.0-0.3	Very Low

LLM-as-Judge System

The LLM-as-judge approach represents the frontier of evaluation methodology, leveraging the reasoning capabilities of advanced language models to assess outputs that resist traditional automated scoring. Think of this as recruiting an expert human evaluator who never gets tired, maintains consistent standards, and can process thousands of evaluations per hour.

This approach excels in scenarios where rule-based metrics fail: evaluating creativity, assessing argument quality, checking factual accuracy, or measuring adherence to complex style guidelines. However, it introduces new challenges around consistency, cost, and potential bias that require careful system design.

Judge Model Selection and Configuration

The effectiveness of LLM-as-judge evaluation depends heavily on selecting appropriate judge models and configuring them for consistent, reliable scoring. Different judge models excel in different evaluation domains.

Judge Model	Strengths	Optimal Use Cases	Consistency Rating
GPT-4	Strong reasoning, broad knowledge	Complex reasoning, factual accuracy	High
Claude-3-Opus	Careful analysis, nuanced judgment	Creative writing, ethical reasoning	High
GPT-3.5-turbo	Fast, cost-effective	Basic quality assessment, format checking	Medium
Llama-2-70B	Open source, customizable	Domain-specific evaluation with fine-tuning	Medium

The judge configuration includes several critical parameters that significantly impact evaluation quality:

Parameter	Recommended Value	Impact on Evaluation	Tuning Strategy
temperature	0.1-0.2	Lower values increase consistency	Start at 0.1, increase if responses too rigid
max_tokens	200-500	Allows detailed explanations	Balance between explanation quality and cost
top_p	0.9	Controls response diversity	Keep high to allow nuanced reasoning
frequency_penalty	0.0	Prevent repetitive language	Usually unnecessary for evaluation tasks

Prompt Engineering for Consistent Evaluation

The quality of LLM-as-judge evaluation depends critically on prompt design. Effective evaluation prompts must provide clear criteria, consistent formatting, and appropriate examples while avoiding bias or leading the judge toward particular scores.

Our prompt template follows a structured format:

1. **Role definition:** Establish the judge's expertise and evaluation perspective
2. **Criteria specification:** Define specific, measurable evaluation dimensions
3. **Scoring rubric:** Provide clear numerical scale with behavioral anchors
4. **Output format:** Specify exact format for score and explanation
5. **Examples:** Include 2-3 golden examples showing excellent evaluation

Consistency Control Mechanisms

LLM judges can exhibit inconsistency across evaluation runs due to their probabilistic nature. We implement several mechanisms to improve consistency and detect when judge reliability degrades.

Consistency Mechanism	Implementation	Reliability Improvement	Performance Impact
Multi-judge consensus	Run 3-5 judges, take median score	High - reduces individual judge variance	3-5x compute cost
Repeated evaluation	Same judge evaluates twice, flag large differences	Medium - detects unstable judgments	2x compute cost
Calibration examples	Include reference cases in each prompt	Medium - anchors judge expectations	10-20% prompt length increase
Confidence scoring	Request confidence level with each score	Low - helps identify uncertain cases	Minimal

Critical Design Decision: We default to single-judge evaluation with calibration examples for cost efficiency, but enable multi-judge consensus for high-stakes evaluations where accuracy justifies additional expense.

Judge Prompt Template System

Our system supports multiple prompt templates optimized for different evaluation scenarios. Each template has been refined through extensive testing to maximize inter-judge agreement and correlation with human expert evaluation.

Template Type	Evaluation Focus	Prompt Structure	Validation Method
accuracy.Focused	Factual correctness, logical consistency	Claim verification checklist	Expert fact-checker agreement
creativity.Focused	Originality, creative expression	Multi-dimensional creativity rubric	Human creativity assessment correlation
helpfulness.Focused	User utility, practical value	Task completion effectiveness	User satisfaction surveys
safety.Focused	Harmful content, bias detection	Risk assessment framework	Safety expert review

Bias Detection and Mitigation

LLM judges can exhibit systematic biases that skew evaluation results. Our framework includes automated bias detection and mitigation strategies to ensure fair evaluation across different content types and demographic groups.

Common bias patterns we monitor include:

1. **Length bias:** Systematically preferring longer or shorter responses
2. **Style bias:** Favoring particular writing styles or formats
3. **Content bias:** Higher scores for certain topics or viewpoints
4. **Demographic bias:** Different scoring patterns for content mentioning different groups
5. **Recency bias:** Score drift over long evaluation runs

Error Handling and Fallback Strategies

LLM-as-judge evaluation introduces new failure modes that require robust error handling. The system must gracefully handle API failures, malformed responses, and judge responses that don't follow the specified format.

Error Type	Detection Method	Fallback Strategy	Recovery Action
API timeout	Request timeout after 30s	Retry with exponential backoff	Queue for later retry
Malformed score	Regex validation of response	Extract numeric values, flag for review	Use partial score if extractable
Missing explanation	Check for explanation field	Request explanation in follow-up call	Mark explanation as missing
Inconsistent scoring	Compare with score distribution	Flag as outlier for manual review	Include in analysis with warning
Rate limiting	429 HTTP status code	Exponential backoff with jitter	Automatic retry after delay

Custom Metric Plugin System

The plugin system enables users to extend our evaluation framework with domain-specific metrics while maintaining the consistency and reliability of our core architecture. Think of this as a standardized electrical outlet system - plugins can provide vastly different functionality (powering a lamp vs. a computer) while conforming to a universal interface that ensures safe, predictable operation.

Plugin Architecture Design

The plugin system balances flexibility for metric developers with safety and performance requirements for production evaluation pipelines. Plugins operate within a controlled sandbox environment that prevents system compromise while providing access to necessary evaluation resources.

Decision: Sandboxed Plugin Execution

- **Context:** Custom metrics might contain bugs, security vulnerabilities, or resource leaks
- **Options Considered:**
 1. Direct in-process execution
 2. Subprocess isolation with resource limits
 3. Container-based sandboxing
- **Decision:** Subprocess isolation with resource limits
- **Rationale:** Balances security with performance - prevents system compromise while avoiding container overhead
- **Consequences:** Some performance overhead but robust isolation and resource control

Plugin Registration and Discovery

The `MetricRegistry` provides a centralized system for plugin discovery, validation, and lifecycle management. Plugins can be registered programmatically, loaded from configuration files, or discovered automatically from designated directories.

Registration Method	Use Case	Validation Level	Loading Time
Programmatic	Built-in metrics, test scenarios	Full validation	Immediate
Configuration file	Deployment-specific metrics	Schema validation	Application startup
Directory scanning	Development, experimentation	Runtime validation	First use
Package installation	Third-party metric libraries	Package signature validation	Import time

The registration process includes comprehensive validation to catch common plugin development errors before they impact evaluation runs:

1. **Interface compliance:** Verify all required methods are implemented
2. **Score range validation:** Confirm scores stay within 0.0-1.0 bounds
3. **Resource usage limits:** Check memory and CPU consumption on sample inputs
4. **Error handling:** Verify graceful handling of malformed inputs
5. **Thread safety:** Validate safe concurrent execution

Plugin Development Framework

To simplify custom metric development, we provide a comprehensive development framework with templates, testing utilities, and debugging tools. Plugin developers can focus on their scoring logic rather than infrastructure concerns.

Basic Plugin Template Structure:

Component	Purpose	Developer Responsibility	Framework Provides
Scoring logic	Core metric computation	Implement domain-specific algorithm	Input validation, error handling
Applicability rules	Test case filtering	Define when metric applies	Tag matching utilities, metadata access
Configuration	Runtime parameters	Define tunable parameters	Parameter validation, type checking
Testing	Validation and regression	Provide test cases	Test runner, performance benchmarks

Plugin Resource Management

Custom metrics run within controlled resource limits to prevent runaway plugins from impacting system stability. The resource management system monitors and enforces limits on memory usage, CPU time, and external API calls.

Resource Type	Default Limit	Monitoring Method	Enforcement Action
Memory usage	100MB per metric	Process memory tracking	Kill subprocess, mark as failed
CPU time	10 seconds per evaluation	Process CPU time	Interrupt execution, return timeout error
API calls	5 calls per evaluation	HTTP request interception	Block additional calls, log warning
File system	Read-only access	System call filtering	Deny write operations
Network access	Configurable whitelist	Network policy enforcement	Block unauthorized connections

Plugin Testing and Validation Framework

The framework provides comprehensive testing utilities that help plugin developers create reliable, consistent metrics. The testing system validates both functional correctness and non-functional requirements like performance and thread safety.

Test Category	Validation Focus	Automated Checks	Manual Verification
Functional	Correct score computation	Reference input/output pairs	Domain expert review
Performance	Resource usage, latency	Benchmark against limits	Scalability testing
Reliability	Error handling, edge cases	Malformed input testing	Stress testing
Consistency	Score stability across runs	Multiple evaluation runs	Inter-rater reliability

Common Pitfalls

⚠ Pitfall: Score Range Violations Custom metrics sometimes return scores outside the required 0.0-1.0 range, breaking aggregation logic. This typically happens when developers use raw similarity scores or count-based metrics without normalization. The framework validates score ranges and raises exceptions for violations, but plugin developers should implement explicit normalization in their metrics rather than relying on framework validation.

⚠ Pitfall: Semantic Similarity Threshold Misunderstanding Many developers assume that high cosine similarity scores (like 0.8) automatically indicate strong semantic similarity, but embedding models have different score distributions.

A 0.8 similarity might represent strong similarity in one model but weak similarity in another. Always calibrate thresholds using golden examples from your specific domain and embedding model combination.

⚠ Pitfall: LLM Judge Prompt Instability Small changes in judge prompts can cause dramatic score distribution shifts that appear as false performance regressions. This happens because LLMs are highly sensitive to prompt wording, format, and example selection. Implement prompt versioning, validate prompts on calibration datasets before production use, and monitor judge score distributions for unexpected shifts.

⚠ Pitfall: Preprocessing Order Dependency The order of text preprocessing steps significantly affects final scores, but many developers treat preprocessing as a simple pipeline. For example, removing punctuation before lowercasing produces different results than lowercasing before removing punctuation. Make preprocessing order explicit and consistent across metrics, and test different orderings on your specific data to identify the most appropriate sequence.

⚠ Pitfall: Plugin Resource Leaks Custom metrics that make external API calls or load large models often fail to clean up resources properly, leading to memory leaks or connection pool exhaustion during long evaluation runs. Always implement proper resource cleanup in plugin code using context managers or try/finally blocks, and test plugins under extended execution scenarios.

⚠ Pitfall: Metric Applicability Over-Filtering Overly restrictive applicability rules can result in test cases being evaluated by too few metrics, reducing the comprehensiveness of evaluation. This often happens when developers create highly specific tag requirements. Prefer inclusive applicability rules and let aggregation handle metric diversity rather than restricting which metrics can evaluate which test cases.

Implementation Guidance

A. Technology Recommendations

Component	Simple Option	Advanced Option
Embedding computation	<code>sentence-transformers</code> library	OpenAI Embeddings API with caching
Text preprocessing	Basic string operations (<code>str.lower()</code> , <code>re.sub()</code>)	<code>spaCy</code> with custom preprocessing pipelines
LLM API integration	<code>openai</code> client library	<code>langchain</code> with retry logic and rate limiting
Plugin system	Direct Python imports	<code>importlib</code> with subprocess isolation
BLEU/ROUGE calculation	<code>nltk.translate.bleu_score</code>	<code>evaluate</code> library from HuggingFace
Similarity computation	<code>sklearn.metrics.pairwise.cosine_similarity</code>	Optimized NumPy operations with batch processing

B. Recommended File Structure

```
metrics_engine/
    __init__.py           ← Public API exports
    base.py               ← BaseMetric abstract class
    registry.py           ← MetricRegistry implementation
    built_in/
        __init__.py
        exact_match.py    ← ExactMatchMetric implementation
        bleu_rouge.py     ← BLEU and ROUGE metrics
        semantic_similarity.py  ← Embedding-based similarity
        llm_judge.py      ← LLM-as-judge implementation
    plugins/
        __init__.py
        plugin_manager.py ← Plugin loading and validation
        resource_limiter.py ← Resource usage enforcement
    templates/
        custom_metric_template.py ← Template for new plugins
utils/
    preprocessing.py      ← Text normalization utilities
    embedding_cache.py    ← Embedding computation and caching
    prompt_templates.py  ← LLM judge prompt management
tests/
    test_base_metric.py
    test_builtin_metrics.py
    test_plugin_system.py
fixtures/
    golden_examples.json ← Reference cases for validation
```

C. Infrastructure Starter Code

Complete Text Preprocessing Utilities:

```
"""

Text preprocessing utilities for consistent metric computation.

Copy this file and use these functions in your metric implementations.

"""

import re

import unicodedata

from typing import List, Optional

from dataclasses import dataclass

@dataclass

class PreprocessingConfig:

    normalize_whitespace: bool = True

    normalize_case: bool = True

    remove_punctuation: bool = False

    normalize_unicode: bool = True

    strip_html: bool = False

    custom_patterns: Optional[List[tuple]] = None # (pattern, replacement) pairs

class TextPreprocessor:

    """Complete text preprocessing with configurable steps."""

    def __init__(self, config: PreprocessingConfig):

        self.config = config

        self._html_pattern = re.compile(r'<[^>]+>')

        self._whitespace_pattern = re.compile(r'\s+')

        self._punctuation_pattern = re.compile(r'[^w\s]')

    def preprocess(self, text: str) -> tuple[str, List[str]]:

        """

        Preprocess text and return (processed_text, applied_steps).

        applied_steps tracks what transformations were applied.

        """
```

```
"""

if not text:

    return "", []


result = text
applied_steps = []


if self.config.strip_html:

    result = self._html_pattern.sub(' ', result)

    applied_steps.append("strip_html")


if self.config.normalize_unicode:

    result = unicodedata.normalize('NFKC', result)

    applied_steps.append("normalize_unicode")


if self.config.normalize_whitespace:

    result = self._whitespace_pattern.sub(' ', result).strip()

    applied_steps.append("normalize_whitespace")


if self.config.normalize_case:

    result = result.lower()

    applied_steps.append("normalize_case")


if self.config.remove_punctuation:

    result = self._punctuation_pattern.sub(' ', result)

    result = self._whitespace_pattern.sub(' ', result).strip()

    applied_steps.append("remove_punctuation")


if self.config.custom_patterns:
```

```

        for pattern, replacement in self.config.custom_patterns:

            result = re.sub(pattern, replacement, result)

            applied_steps.append(f"custom_pattern_{pattern}")



    return result, applied_steps

# Ready-to-use preprocessor instances

STRICT_PREPROCESSOR = TextPreprocessor(PreprocessingConfig(
    normalize_whitespace=False,
    normalize_case=False,
    remove_punctuation=False,
    normalize_unicode=False
))

NORMALIZED_PREPROCESSOR = TextPreprocessor(PreprocessingConfig(
    normalize_whitespace=True,
    normalize_case=True,
    remove_punctuation=False,
    normalize_unicode=True
))

FUZZY_PREPROCESSOR = TextPreprocessor(PreprocessingConfig(
    normalize_whitespace=True,
    normalize_case=True,
    remove_punctuation=True,
    normalize_unicode=True,
    strip_html=True
))

```

Complete Embedding Cache System:

```
"""

Embedding computation and caching system.

Handles multiple embedding providers with automatic caching and batch processing.

"""

import hashlib

import pickle

import sqlite3

from pathlib import Path

from typing import List, Optional, Dict, Any

import numpy as np

from sentence_transformers import SentenceTransformer

import openai

class EmbeddingCache:

    """Persistent cache for embedding computations."""

    def __init__(self, cache_dir: Path):

        self.cache_dir = cache_dir

        self.cache_dir.mkdir(exist_ok=True)

        self.db_path = cache_dir / "embeddings.db"

        self._init_database()

    def _init_database(self):

        """Initialize SQLite database for embedding storage."""

        conn = sqlite3.connect(self.db_path)

        conn.execute("""

            CREATE TABLE IF NOT EXISTS embeddings (

                text_hash TEXT PRIMARY KEY,

                model_name TEXT NOT NULL,

                embedding BLOB NOT NULL,
        
```



```
        (text_hash, model_name, pickle.dumps(embedding))

    )

    conn.commit()

    conn.close()

class EmbeddingProvider:

    """Unified interface for different embedding providers."""

    def __init__(self, cache_dir: Path):

        self.cache = EmbeddingCache(cache_dir)

        self._sentence_transformer = None

        self._openai_client = None

    def get_embeddings(self, texts: List[str], provider: str, model: str) -> np.ndarray:

        """Get embeddings for list of texts, using cache when possible."""

        embeddings = []

        uncached_texts = []

        uncached_indices = []

        # Check cache first

        for i, text in enumerate(texts):

            cached_embedding = self.cache.get_embedding(text, f"{provider}:{model}")

            if cached_embedding is not None:

                embeddings.append(cached_embedding)

            else:

                embeddings.append(None)

                uncached_texts.append(text)

                uncached_indices.append(i)

        # Compute missing embeddings
```

```
if uncached_texts:

    if provider == "sentence_transformers":

        new_embeddings = self._compute_sentence_transformer_embeddings(uncached_texts, model)

    elif provider == "openai":

        new_embeddings = self._compute_openai_embeddings(uncached_texts, model)

    else:

        raise ValueError(f"Unknown embedding provider: {provider}")

    # Store in cache and update results

    for i, (text, embedding) in enumerate(zip(uncached_texts, new_embeddings)):

        self.cache.store_embedding(text, f"{provider}:{model}", embedding)

        embeddings[uncached_indices[i]] = embedding

return np.array(embeddings)

def _compute_sentence_transformer_embeddings(self, texts: List[str], model: str) -> np.ndarray:

    """Compute embeddings using SentenceTransformers."""

    if self._sentence_transformer is None or self._sentence_transformer.model_name != model:

        self._sentence_transformer = SentenceTransformer(model)

    return self._sentence_transformer.encode(texts)

def _compute_openai_embeddings(self, texts: List[str], model: str) -> np.ndarray:

    """Compute embeddings using OpenAI API."""

    if self._openai_client is None:

        self._openai_client = openai.OpenAI() # Uses OPENAI_API_KEY env var

    response = self._openai_client.embeddings.create(
        input=texts,
```

```
    model=model

    )

embeddings = [data.embedding for data in response.data]

return np.array(embeddings)
```

D. Core Logic Skeleton Code

BaseMetric Abstract Class:

```
from abc import ABC, abstractmethod

from typing import Dict, Any, List, Optional

from dataclasses import dataclass

import time

@dataclass

class MetricResult:

    score: float

    metadata: Dict[str, Any]

    explanation: Optional[str] = None

    computation_time: float = 0.0

    error_info: Optional[str] = None

    preprocessing_applied: List[str] = None


    def __post_init__(self):

        if self.preprocessing_applied is None:

            self.preprocessing_applied = []


class BaseMetric(ABC):

    """


    Abstract base class for all evaluation metrics.

    Provides consistent interface and common functionality.

    """


    def __init__(self, name: str, description: str):

        self.name = name

        self.description = description


    @abstractmethod

    def compute(self, model_output: str, expected_output: str, test_case: 'TestCase') ->

MetricResult:
```

```
"""
```

```
Compute metric score for given model output vs expected output.
```

```
Args:
```

```
    model_output: The text produced by the model being evaluated  
    expected_output: The reference/ground truth output  
    test_case: Complete test case with metadata and tags
```

```
Returns:
```

```
    MetricResult with score (0.0-1.0), metadata, and explanation
```

```
"""
```

```
# TODO 1: Validate inputs - check for None/empty strings  
  
# TODO 2: Apply preprocessing if needed (call self.get_preprocessing_steps())  
  
# TODO 3: Implement core scoring logic  
  
# TODO 4: Normalize score to 0.0-1.0 range  
  
# TODO 5: Create explanation of how score was computed  
  
# TODO 6: Return MetricResult with all fields populated  
  
pass
```

```
@abstractmethod
```

```
def is_applicable(self, test_case: 'TestCase') -> bool:
```

```
"""
```

```
Determine if this metric should be applied to the given test case.
```

```
Args:
```

```
    test_case: Test case to check applicability for
```

```
Returns:
```

```
    True if metric should run on this test case, False otherwise
```

```
"""

# TODO 1: Check test_case.tags for required tags

# TODO 2: Check test_case.difficulty if metric has difficulty requirements

# TODO 3: Check test_case.metadata for any custom requirements

# TODO 4: Return boolean decision

pass


def get_name(self) -> str:
    """Return unique identifier for this metric type."""
    return self.name


def get_description(self) -> str:
    """Return human-readable description of what this metric measures."""
    return self.description


def requires_preprocessing(self) -> bool:
    """Override if metric needs text preprocessing before scoring."""
    return False


def get_preprocessing_steps(self) -> List[str]:
    """Override to specify preprocessing operations needed."""
    return []


def _safe_compute(self, model_output: str, expected_output: str, test_case: 'TestCase') -> MetricResult:
    """
    Wrapper that adds timing and error handling to compute method.

    Framework calls this instead of compute() directly.
    """
    start_time = time.time()
```

```
try:

    # TODO 1: Call self.compute() with provided arguments

    # TODO 2: Validate returned MetricResult has score in 0.0-1.0 range

    # TODO 3: Add computation_time to result

    # TODO 4: Return validated result

    pass

except Exception as e:

    # TODO 5: Create MetricResult with score=0.0 and error_info

    # TODO 6: Log error for debugging

    # TODO 7: Return error result

    pass
```

Exact Match Metric Implementation:

```
class ExactMatchMetric(BaseMetric):  
    """  
    Exact string matching with configurable normalization options.  
    Supports strict, normalized, and fuzzy matching modes.  
    """  
  
    def __init__(self, matching_mode: str = "normalized"):  
        super().__init__(  
            name=f"exact_match_{matching_mode}",  
            description=f"Exact string matching with {matching_mode} preprocessing"  
        )  
        self.matching_mode = matching_mode  
  
        if matching_mode == "strict":  
            self.preprocessor = STRICT_PREPROCESSOR  
        elif matching_mode == "normalized":  
            self.preprocessor = NORMALIZED_PREPROCESSOR  
        elif matching_mode == "fuzzy":  
            self.preprocessor = FUZZY_PREPROCESSOR  
        else:  
            raise ValueError(f"Unknown matching mode: {matching_mode}")  
  
    def compute(self, model_output: str, expected_output: str, test_case: 'TestCase') ->  
        MetricResult:  
        # TODO 1: Handle None/empty input cases - return score 0.0 with explanation  
        # TODO 2: Apply preprocessing using self.preprocessor.preprocess()  
        # TODO 3: Compare preprocessed strings with == operator  
        # TODO 4: If exact match, score = 1.0; otherwise score = 0.0  
        # TODO 5: Check if substring matching is enabled in test_case.metadata  
        # TODO 6: If substring matching: check if expected_output in model_output
```

```

# TODO 7: Create explanation describing match result and preprocessing applied

# TODO 8: Return MetricResult with score, explanation, and preprocessing_applied list

pass


def is_applicable(self, test_case: 'TestCase') -> bool:

    # TODO 1: Return True if "exact_match" in test_case.tags

    # TODO 2: Also return True if test_case.difficulty == "easy" (exact match good for simple
    cases)

    # TODO 3: Check test_case.metadata for "disable_exact_match" flag, return False if present

    # TODO 4: Default to True - exact match is broadly applicable

    pass


def requires_preprocessing(self) -> bool:

    return self.matching_mode != "strict"


def get_preprocessing_steps(self) -> List[str]:

    if self.matching_mode == "strict":

        return []

    elif self.matching_mode == "normalized":

        return ["normalize_whitespace", "normalize_case", "normalize_unicode"]

    elif self.matching_mode == "fuzzy":

        return ["normalize_whitespace", "normalize_case", "remove_punctuation",
    "normalize_unicode"]

```

Semantic Similarity Metric Implementation:

```
class SemanticSimilarityMetric(BaseMetric):  
    """  
    Embedding-based semantic similarity using cosine distance.  
    Supports multiple embedding providers with automatic caching.  
    """  
  
    def __init__(self, provider: str = "sentence_transformers",  
                 model: str = "all-MiniLM-L6-v2",  
                 cache_dir: Path = Path("./embedding_cache")):  
        super().__init__(  
            name=f"semantic_similarity_{provider}_{model.replace('/', '_')}",  
            description=f"Semantic similarity using {provider} {model}"  
        )  
        self.provider = provider  
        self.model = model  
        self.embedding_provider = EmbeddingProvider(cache_dir)  
  
        # Calibration thresholds - tune these based on your domain  
        self.thresholds = {  
            "very_high": 0.95,  
            "high": 0.85,  
            "medium": 0.70,  
            "low": 0.55  
        }  
  
    def compute(self, model_output: str, expected_output: str, test_case: 'TestCase') ->  
    MetricResult:  
        # TODO 1: Handle empty/None inputs - return 0.0 score  
        # TODO 2: Apply basic preprocessing (normalize whitespace, strip)  
        # TODO 3: Get embeddings for both texts using self.embedding_provider.get_embeddings()
```

```

# TODO 4: Compute cosine similarity between embeddings using sklearn or numpy

# TODO 5: Apply calibration mapping using self._calibrate_similarity_score()

# TODO 6: Create explanation describing similarity level and confidence

# TODO 7: Add raw_similarity and calibrated_score to metadata

# TODO 8: Return MetricResult with calibrated score and detailed metadata

pass

def is_applicable(self, test_case: 'TestCase') -> bool:

    # TODO 1: Return True if "semantic" in test_case.tags

    # TODO 2: Return True if "reasoning" in test_case.tags (semantic similarity good for
    reasoning tasks)

    # TODO 3: Return False if "code" in test_case.tags (semantic similarity poor for code)

    # TODO 4: Return False if test_case.difficulty == "easy" (exact match better for simple
    cases)

    # TODO 5: Check expected_output length - return False if very short (< 10 words)

    pass

def _calibrate_similarity_score(self, raw_similarity: float) -> tuple[float, str]:
    """
    Map raw cosine similarity to calibrated 0.0-1.0 score with confidence level.

    Override this method to customize calibration for your domain.

    """
    # TODO 1: Compare raw_similarity against self.thresholds

    # TODO 2: Map to appropriate score range based on threshold category

    # TODO 3: Return (calibrated_score, confidence_level_string)

    # Hint: Very high similarity (>0.95) maps to 1.0, low similarity (<0.55) maps to 0.0-0.3

    pass

```

E. Language-Specific Hints

- **Embedding computation:** Use `sentence-transformers` for fast local embeddings or OpenAI API for higher quality. Always implement caching to avoid recomputing embeddings for the same text.

- **Cosine similarity:** Use `sklearn.metrics.pairwise.cosine_similarity()` or implement with NumPy:
`np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))`
- **LLM API calls:** Always implement retry logic with exponential backoff using `tenacity` library. Set reasonable timeouts (30-60 seconds) for judge evaluations.
- **Text preprocessing:** Use `re.sub(r'\s+', ' ', text).strip()` for whitespace normalization. Consider `unicodedata.normalize('NFKC', text)` for Unicode normalization.
- **Plugin system:** Use `importlib.import_module()` for dynamic loading. Implement resource limits with `resource.setrlimit()` and subprocess execution.
- **Score validation:** Always validate that metric scores are in [0.0, 1.0] range: `assert 0.0 <= score <= 1.0, f"Invalid score: {score}"`

F. Milestone Checkpoint

After implementing the metrics engine:

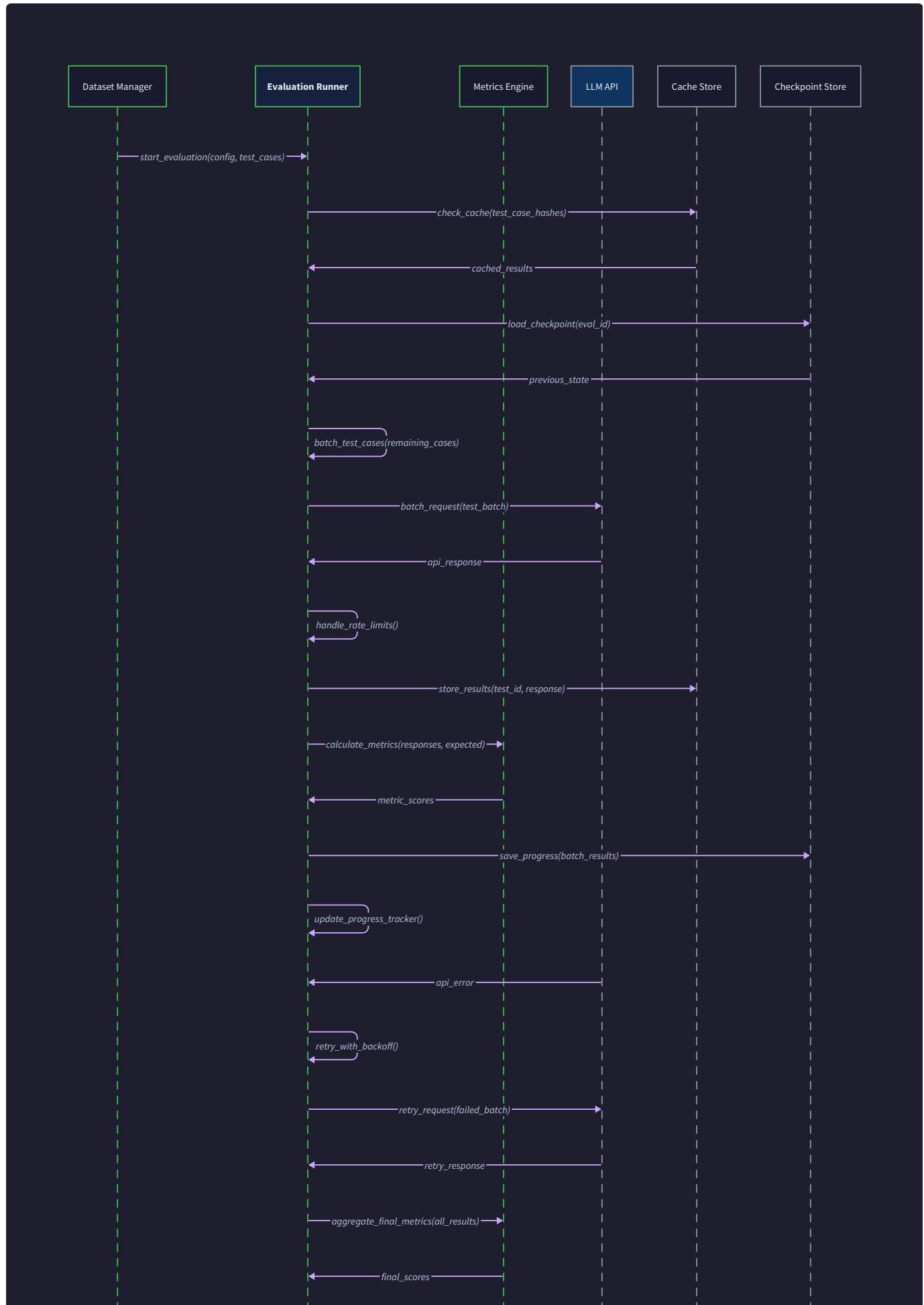
- **Test command:** `python -m pytest tests/test_metrics_engine.py -v`
- **Expected output:** All built-in metrics pass validation tests, custom metric registration works, LLM judge produces consistent scores
- **Manual verification:**
 1. Create a simple test case: `prompt="What is 2+2?", expected="4"`
 2. Run exact match metric - should score 1.0 for exact match, 0.0 for "four"
 3. Run semantic similarity - should score high (>0.8) for "four", medium for "2 plus 2 equals 4"
 4. Run LLM judge with creativity prompt - should provide detailed explanation with score
- **Performance check:** Metrics should process 100 test cases in under 30 seconds (excluding LLM API calls)
- **Signs of problems:**
 - Scores outside 0.0-1.0 range indicate normalization bugs
 - High variance in repeated LLM judge evaluations suggests prompt instability
 - Memory leaks during large batch processing indicate resource cleanup issues

Evaluation Runner

Milestone(s): Milestone 3 (Evaluation Runner) - This section covers the complete evaluation execution system including batch processing, concurrency control, result caching, checkpointing, and progress tracking

The evaluation runner is the orchestration engine that coordinates the execution of large-scale evaluations across thousands of test cases. Think of it as a sophisticated job scheduler for a distributed computing system - it needs to balance throughput, reliability, and resource efficiency while providing visibility into progress and handling inevitable failures gracefully.

The core challenge is managing the inherent unpredictability of LLM API calls. Unlike traditional batch processing where each task has predictable execution time and failure modes, LLM evaluation involves network requests with variable latency, rate limits that change dynamically, and API responses that may be malformed or incomplete. The evaluation runner must handle these uncertainties while maintaining high throughput and providing reliable progress estimates.



```
evaluation_complete(results, metrics)
```

The evaluation runner operates on a four-layer architecture: the execution engine manages batch processing and concurrency, the caching system eliminates redundant API calls, the checkpoint system ensures crash recovery, and the progress tracking system provides real-time visibility. Each layer has distinct responsibilities but must coordinate closely to achieve optimal performance.

Execution Engine

The execution engine is the heart of the evaluation runner, responsible for orchestrating parallel evaluation across potentially thousands of test cases while respecting API rate limits and resource constraints. Think of it as an air traffic control system - it needs to coordinate many concurrent operations safely while maximizing overall throughput.

The fundamental challenge is balancing concurrency for speed against stability and rate limiting. Too much concurrency overwhelms APIs and causes cascade failures. Too little concurrency leaves performance on the table. The execution engine dynamically adjusts concurrency based on observed API response times and error rates.

Decision: Adaptive Batch Processing Architecture

- **Context:** Large evaluation datasets (10K+ test cases) require parallel processing for reasonable completion times, but LLM APIs have complex rate limiting and reliability characteristics
- **Options Considered:** Fixed batch sizes, dynamic batch sizing based on API latency, per-provider adaptive concurrency
- **Decision:** Implement adaptive batch processing with per-provider concurrency control and exponential backoff
- **Rationale:** Fixed approaches fail under varying API conditions. Adaptive sizing allows the system to automatically optimize for current API performance while preventing overload
- **Consequences:** Enables high throughput under good conditions while maintaining stability during API degradation, but requires more complex monitoring and tuning logic

Component	Responsibility	Key Interfaces	Error Handling
BatchProcessor	Divides test cases into optimally-sized batches	<code>process_batch(items, processor_func)</code>	Retries failed batches with exponential backoff
ConcurrencyManager	Controls parallel evaluation threads per API provider	<code>acquire_slot(provider)</code> , <code>release_slot(provider)</code>	Blocks new requests when provider error rate exceeds threshold
RateLimiter	Enforces API rate limits to prevent rejection	<code>acquire()</code> , <code>get_wait_time()</code>	Implements token bucket algorithm with provider-specific limits
ProviderRouter	Routes evaluation requests to appropriate LLM providers	<code>route_request(test_case)</code> , <code>get_healthy_providers()</code>	Fails over to backup providers when primary is unavailable

The execution engine processes evaluations in three phases: preparation, execution, and aggregation. During preparation, it analyzes the dataset to determine optimal batching strategy based on test case complexity and estimated execution time. The execution phase processes batches in parallel while continuously monitoring API health and adjusting concurrency. The aggregation phase combines results from all batches and computes final statistics.

Batch Size Optimization Algorithm:

1. The system starts with a conservative baseline batch size (typically 10-20 test cases) based on the configured concurrency limits
2. It measures the actual processing time for each completed batch, including API latency, metric computation, and result serialization
3. For each LLM provider, it maintains a rolling average of successful batch completion times over the last 50 batches
4. If the average completion time is below the target threshold (e.g., 30 seconds), the system increases batch size by 25%
5. If completion times exceed the threshold or error rates rise above 5%, batch size is reduced by 50%
6. The system maintains separate batch size settings for each provider since their performance characteristics differ significantly
7. Batch sizes are bounded between a minimum of 5 (to handle degraded conditions) and maximum of 100 (to prevent excessive memory usage)

The concurrency manager implements a sophisticated slot-based system where each LLM provider has a dedicated pool of execution slots. This prevents one slow provider from blocking evaluation progress with other providers. The slot allocation is dynamic - providers with better performance characteristics receive more slots during peak processing.

The critical insight is that batch processing for LLM evaluation must be adaptive rather than static. API performance varies dramatically based on time of day, model load, and provider infrastructure changes. A system optimized for peak performance will crash during degraded conditions, while one optimized for reliability will be unnecessarily slow during good conditions.

Provider Health Monitoring:

The execution engine continuously monitors each LLM provider's health using multiple metrics. Response latency is tracked using exponential moving averages with recent responses weighted more heavily. Error rates are computed over sliding windows to detect both sudden failures and gradual degradation. The system maintains separate health scores for different types of errors - transient network issues are treated differently from authentication failures or rate limit rejections.

Health Metric	Measurement Window	Healthy Threshold	Degraded Threshold	Recovery Criteria
Response Latency	50 requests	< 5 seconds	> 15 seconds	20 consecutive requests < 8 seconds
Error Rate	100 requests	< 2%	> 10%	Error rate < 1% for 5 minutes
Rate Limit Hits	10 minutes	< 1 per minute	> 5 per minute	No rate limits for 15 minutes
Timeout Rate	50 requests	< 5%	> 20%	Timeout rate < 3% for 10 minutes

When a provider enters degraded state, the execution engine reduces its concurrency allocation and begins routing new requests to healthy providers. If all providers are degraded, the system enters backoff mode where it processes requests sequentially with increasing delays between attempts.

Result Caching System

The result caching system eliminates redundant API calls by storing LLM responses keyed by prompt content and provider configuration. Think of it as a sophisticated memoization layer - it remembers previous computations so identical requests can be answered instantly without expensive API calls.

The challenge with caching LLM responses is determining when two prompts are "identical" enough to return cached results. Simple string equality is too restrictive since minor formatting differences shouldn't invalidate the cache. However, semantic equivalence is too expensive to compute at query time. The caching system uses a hybrid approach with content-based hashing and configurable normalization rules.

Decision: Content-Addressable Cache with Layered Keys

- **Context:** LLM API calls are expensive (time and cost) and many evaluation runs contain duplicate or near-duplicate test cases
- **Options Considered:** Simple prompt hash, semantic similarity clustering, multi-level cache hierarchy
- **Decision:** Content-addressable storage using normalized prompt hash as primary key with provider config hash as secondary key
- **Rationale:** Provides exact match caching with predictable performance while allowing cache hits across different evaluation runs
- **Consequences:** Eliminates redundant API calls for identical prompts but misses opportunities for semantic similarity caching

Cache Component	Purpose	Key Structure	Storage Backend	TTL Policy
PromptCache	Stores LLM responses by prompt content	<code>sha256(normalized_prompt + provider_config)</code>	SQLite with WAL mode	30 days for successful responses
EmbeddingCache	Caches embeddings for semantic similarity metrics	<code>sha256(text + embedding_model)</code>	HDF5 files with compression	90 days (embeddings more stable)
MetricCache	Caches computed metric results	<code>sha256(model_response + expected_output + metric_config)</code>	Memory-mapped files	Session-only (metrics change frequently)

The prompt normalization process is configurable but typically includes whitespace normalization, consistent line ending conversion, and removal of non-semantic formatting. The normalization must be deterministic and reversible to ensure cache consistency across different execution environments.

Cache Key Generation Algorithm:

1. The system extracts the core prompt content from the `TestCase`, excluding metadata fields like `case_id` and `tags` that don't affect the LLM response
2. It applies configured normalization rules: strip leading/trailing whitespace, normalize line endings to `\n`, collapse multiple consecutive spaces to single spaces
3. The provider configuration is serialized to a canonical JSON representation with keys sorted alphabetically
4. A composite string is formed by concatenating normalized prompt, provider config JSON, and a cache version identifier
5. The final cache key is computed as `sha256(composite_string)` and encoded as a hex string for storage
6. Secondary keys are generated for provider-specific optimizations and cache invalidation patterns

The caching system implements a write-through policy where responses are stored immediately after successful API calls. This ensures cache consistency even if the evaluation process crashes partway through. Cache hits are validated by checking response metadata for completeness and comparing timestamps against configured TTL policies.

Cache Invalidation Strategy:

Cache invalidation is triggered by several events: provider configuration changes, cache version updates, and explicit invalidation requests. The system uses a tombstone approach where invalidated entries are marked as deleted rather than immediately removed, allowing for cache statistics and debugging.

Invalidation Trigger	Scope	Implementation	Recovery Time
Provider Config Change	All entries for affected provider	Mark with tombstone, lazy cleanup	Immediate for new requests
Cache Version Update	Global cache invalidation	Rename cache directory, create fresh	Next evaluation run startup
Explicit Request	User-specified patterns	Pattern matching on cache keys	Immediate
TTL Expiration	Individual entries based on age	Background cleanup process	Continuous

The cache provides detailed statistics including hit rates, storage utilization, and performance impact. These metrics help optimize cache policies and troubleshoot evaluation performance issues.

Checkpoint and Recovery

The checkpoint and recovery system ensures that large evaluation runs can survive crashes, API outages, and other interruptions without losing progress. Think of it as a database transaction log - it records the system's state at regular intervals so recovery can resume from the last known good checkpoint rather than starting over.

The fundamental challenge is balancing checkpoint frequency against performance overhead. Frequent checkpoints minimize lost work but slow down overall execution. Infrequent checkpoints reduce overhead but risk losing significant progress. The system uses adaptive checkpointing that increases frequency during unstable conditions and reduces it during smooth operation.

Decision: Hierarchical Checkpointing with Incremental Updates

- **Context:** Large evaluation runs can take hours to complete, and various failures (API outages, crashes, network issues) can interrupt progress
- **Options Considered:** Full state snapshots at fixed intervals, incremental checkpoints with transaction logs, distributed checkpointing across workers
- **Decision:** Hierarchical checkpointing with fast incremental updates and periodic full snapshots
- **Rationale:** Combines fast recovery (incremental) with guaranteed consistency (full snapshots) while minimizing I/O overhead during normal operation
- **Consequences:** Enables sub-minute recovery times with minimal performance impact, but requires more complex checkpoint management logic

Checkpoint Type	Frequency	Content	Storage Size	Recovery Time
Incremental	Every 10 completed evaluations	Result deltas and progress counters	< 1MB typically	< 10 seconds
Batch Complete	End of each batch	Full batch results and updated statistics	5-50MB per batch	30-60 seconds
Full Snapshot	Every 500 evaluations or 30 minutes	Complete evaluation state	50-500MB	2-5 minutes
Emergency	Before shutdown or on error	Current progress with error context	Variable	< 5 seconds

The checkpoint system maintains a write-ahead log (WAL) of all evaluation state changes. Each entry in the WAL includes a sequence number, timestamp, operation type, and the affected data. This allows the recovery process to replay operations in the correct order and detect any inconsistencies.

Checkpoint State Management:

The checkpointing system tracks multiple categories of state that must be preserved across crashes. Evaluation progress includes completed test cases, pending batches, and provider health metrics. Configuration state includes dataset version, metric configurations, and runtime parameters. Error state includes failed test cases, retry counts, and error classifications.

1. Before starting any evaluation batch, the system writes a `BATCH_START` record to the WAL with batch ID and test case IDs
2. As individual test cases complete, `EVALUATION_COMPLETE` records are written with results and timing information
3. When a full batch completes successfully, a `BATCH_COMPLETE` record summarizes the batch results and updates overall progress
4. Every 500 completed evaluations, a `FULL_SNAPSHOT` record captures the complete evaluation state
5. If an error occurs, an `ERROR_RECORD` captures the failure context and marks affected test cases for retry
6. The WAL is fsynced to disk after each write to ensure durability even during hard crashes
7. Old WAL entries are pruned after successful full snapshots to prevent unbounded growth

The recovery process scans the WAL from the most recent full snapshot forward, replaying operations to reconstruct the evaluation state. It validates each operation for consistency and handles gaps or corruption by falling back to the previous consistent state.

Recovery Process Algorithm:

Recovery begins by scanning the checkpoint directory for the most recent full snapshot and corresponding WAL files. The system validates the snapshot integrity using stored checksums and begins WAL replay from the snapshot's sequence number.

Recovery Phase	Actions Taken	Validation Checks	Fallback Strategy
Snapshot Load	Deserialize evaluation state from snapshot file	Checksum validation, schema version compatibility	Try previous snapshot if current is corrupted
WAL Replay	Apply operations from WAL in sequence order	Sequence number gaps, operation consistency	Stop at first inconsistency, use partial recovery
State Validation	Verify recovered state matches expected invariants	Progress counters, result consistency, provider health	Reset inconsistent components to safe defaults
Resume Preparation	Identify incomplete batches and failed test cases	Check for pending operations, validate retry counts	Mark uncertain test cases for re-evaluation

The key insight for checkpoint recovery is that partial progress is better than no progress. Even if some test cases must be re-evaluated due to uncertainty, preserving the majority of completed work provides significant time savings for large evaluation runs.

Crash Recovery Scenarios:

The checkpoint system handles various failure modes with different recovery strategies. Hard crashes (power loss, kill -9) rely on WAL replay from the last fsynced checkpoint. Soft crashes (exceptions, API failures) can use in-memory state for faster recovery. Network partitions may leave some results in uncertain states that require re-evaluation.

Progress Tracking

The progress tracking system provides real-time visibility into evaluation execution with accurate time estimates and detailed performance metrics. Think of it as a sophisticated dashboard for a complex manufacturing process - it needs to show not just overall progress but also bottlenecks, quality metrics, and predictive estimates for completion.

The challenge with progress tracking for LLM evaluation is that execution time per test case varies dramatically based on prompt complexity, API provider performance, and metric computation requirements. Simple linear extrapolation gives poor time estimates. The tracking system uses statistical modeling based on historical performance to provide accurate completion estimates.

Decision: Multi-Modal Progress Estimation with Confidence Intervals

- **Context:** Users need accurate progress estimates for evaluation runs that can take hours, but LLM API response times are highly variable and unpredictable
- **Options Considered:** Linear extrapolation from average times, exponential smoothing of recent performance, machine learning models based on test case features
- **Decision:** Statistical modeling using test case complexity features and provider performance history with confidence intervals
- **Rationale:** Provides more accurate estimates than simple averages while giving users realistic expectations about estimate uncertainty
- **Consequences:** Requires more sophisticated tracking infrastructure but delivers significantly better user experience for long-running evaluations

Progress Dimension	Measurement Approach	Update Frequency	Accuracy Target
Overall Completion	Count of completed vs total test cases	Real-time	±2% of actual progress
Time Remaining	Statistical model based on test case complexity	Every 10 completions	±20% for 80% of estimates
Provider Performance	Response time percentiles and error rates	Per API call	Real-time health indicators
Quality Metrics	Score distributions and failure analysis	Per completed batch	Batch-level accuracy

The progress tracking system maintains detailed statistics about evaluation performance that inform both real-time estimates and post-evaluation analysis. It tracks not just completion counts but also error patterns, performance bottlenecks, and quality trends.

Complexity-Based Time Estimation:

The progress tracker analyzes each test case to estimate its execution complexity based on multiple factors. Prompt length affects API response time. Expected output length influences metric computation time. The number of applicable metrics determines total scoring overhead. Provider health status affects retry likelihood.

1. For each test case, extract complexity features: prompt token count, expected output token count, number of applicable metrics, provider assignment
2. Look up historical performance data for similar test cases using k-nearest neighbors matching on the feature vector
3. Compute base time estimate as weighted average of similar cases, with more recent examples weighted more heavily
4. Apply provider-specific adjustments based on current health status and observed latency patterns
5. Add metric computation overhead based on the configured metric types (exact match is fast, semantic similarity requires embeddings, LLM-as-judge needs additional API calls)
6. Generate confidence intervals using the variance in historical performance for similar test cases
7. Update estimates continuously as new performance data becomes available

The estimation model is retrained periodically using completed evaluation data to improve accuracy. The system maintains separate models for different providers and metric combinations to capture their distinct performance characteristics.

Real-Time Performance Monitoring:

Progress tracking goes beyond simple completion counts to provide detailed performance insights. The system monitors throughput trends, identifies bottlenecks, and predicts potential issues before they cause significant delays.

Performance Metric	Calculation Method	Normal Range	Alert Threshold	Diagnostic Value
Evaluations/Minute	Rolling average over 5-minute window	Varies by provider	50% below historical average	Identifies throughput degradation
API Response Time	P95 latency over recent 100 requests	< 10 seconds typical	> 30 seconds	Detects provider performance issues
Error Rate	Failed requests / total requests over 10 minutes	< 2%	> 10%	Indicates API or configuration problems
Queue Depth	Pending evaluations waiting for available slots	< 50 typical	> 200	Shows concurrency bottlenecks
Cache Hit Rate	Cache hits / total requests over evaluation run	10-30% typical	< 5%	Indicates caching effectiveness

The monitoring system generates automatic alerts when performance metrics deviate significantly from expected ranges. These alerts include suggested diagnostic actions and potential remediation strategies.

Progress Visualization and Reporting:

The progress tracking system generates multiple views of evaluation progress optimized for different audiences. Developers need detailed technical metrics and error diagnostics. Project managers need high-level completion estimates and risk assessments. The system provides both real-time dashboards and periodic summary reports.

Real-time progress display includes overall completion percentage, estimated time remaining with confidence intervals, current throughput rate, and provider health status. A detailed breakdown shows progress by test case category, metric type, and difficulty level. Error summary highlights the most common failure patterns and affected test cases.

The critical insight for progress tracking is that uncertainty is as important as the estimate itself. Users make better decisions when they understand not just that an evaluation will complete in "2 hours" but that the estimate has a 90% confidence interval of "1.5 to 3 hours" based on current performance trends.

Common Pitfalls:

⚠ Pitfall: Linear Extrapolation for Time Estimates Simple progress calculations like `(total_cases - completed_cases) * average_time_per_case` give misleading estimates because LLM API performance varies dramatically. Early test cases may complete quickly while later complex ones take much longer. Use statistical models based on test case complexity and provider performance history instead.

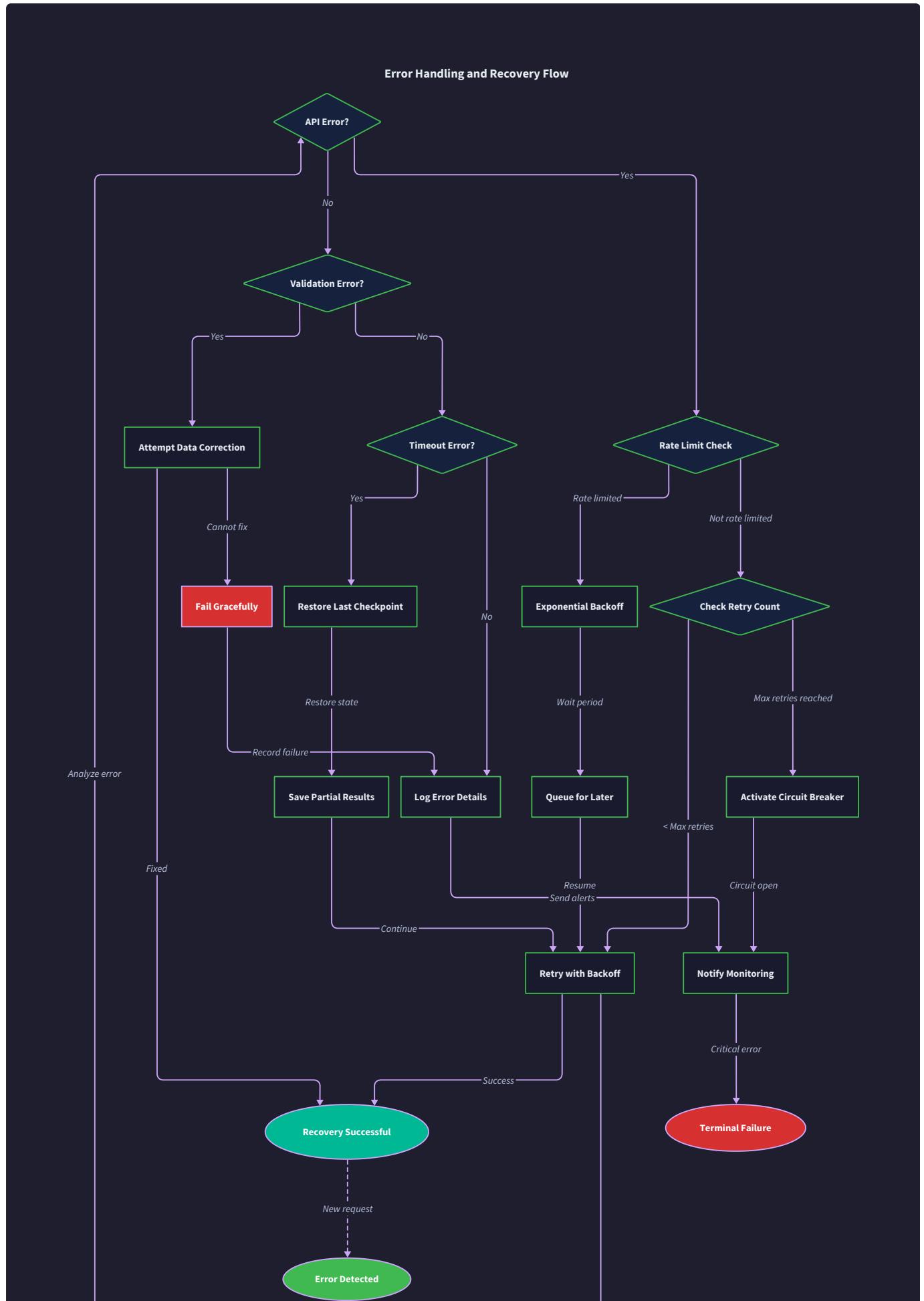
⚠ Pitfall: Ignoring Provider Health in Batch Sizing Fixed batch sizes that work well under normal conditions can cause cascade failures when API providers are degraded. Always incorporate provider health metrics into batch size calculations and reduce concurrency when error rates increase.

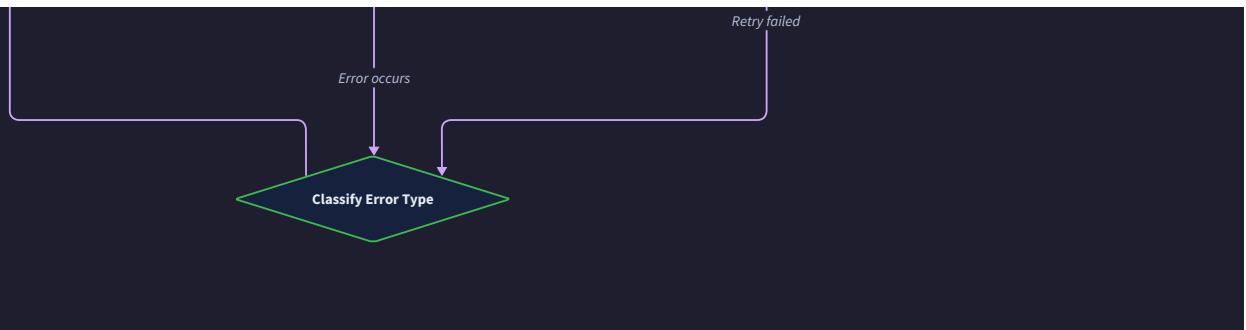
⚠ Pitfall: Cache Keys That Miss Obvious Duplicates Using raw prompt strings as cache keys misses opportunities to cache results for semantically identical prompts with minor formatting differences. Implement prompt normalization that strips non-semantic differences while preserving meaning.

⚠ Pitfall: Checkpoint Corruption During Shutdown Writing checkpoints during system shutdown can result in incomplete or corrupted state files that prevent successful recovery. Use atomic write operations (write to temporary file,

then rename) and validate checkpoint integrity before cleanup.

⚠ Pitfall: Progress Estimates Without Confidence Intervals Giving users precise time estimates like "2 hours 37 minutes remaining" without indicating uncertainty leads to frustration when estimates change. Always provide confidence intervals that reflect the inherent uncertainty in LLM evaluation timing.





Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
Batch Processing	<code>concurrent.futures.ThreadPoolExecutor</code>	<code>asyncio</code> with <code>aiohttp</code> for async API calls
Result Caching	<code>sqlite3</code> with pickle serialization	Redis with JSON serialization and TTL
Progress Tracking	Simple console output with <code>tqdm</code>	Web dashboard with WebSocket updates
Checkpointing	JSON files with atomic writes	SQLite WAL mode with transactions
Rate Limiting	<code>time.sleep()</code> with token bucket	<code>aiohttp_retry</code> with exponential backoff

Recommended File Structure:

```

evaluation_framework/
  evaluation_runner/
    __init__.py
    execution_engine.py      ← BatchProcessor, ConcurrencyManager
    caching.py               ← PromptCache, EmbeddingCache
    checkpointing.py         ← CheckpointManager, WALWriter
    progress_tracking.py     ← ProgressTracker, TimeEstimator
    rate_limiting.py         ← RateLimiter, ProviderHealthMonitor
  tests/
    test_execution_engine.py
    test_caching.py
    test_checkpointing.py
    test_progress_tracking.py

```

Infrastructure Starter Code (Complete Rate Limiter):

```
import time

import threading

from collections import deque

from typing import Dict, Optional

from dataclasses import dataclass


@dataclass

class RateLimitConfig:

    max_requests: int

    time_window: int # seconds

    burst_limit: int = None # allow short bursts


class RateLimiter:

    """Token bucket rate limiter with per-provider limits."""

    def __init__(self, provider_configs: Dict[str, RateLimitConfig]):

        self.provider_configs = provider_configs

        self._tokens: Dict[str, int] = {}

        self._last_refill: Dict[str, float] = {}

        self._request_times: Dict[str, deque] = {}

        self._lock = threading.Lock()

        # Initialize token buckets

        for provider, config in provider_configs.items():

            self._tokens[provider] = config.max_requests

            self._last_refill[provider] = time.time()

            self._request_times[provider] = deque()

    def acquire(self, provider: str) -> bool:

        """Acquire a rate limit token. Returns True if allowed, False if should wait."""

        with self._lock:
```

```
    with self._lock:

        config = self.provider_configs[provider]

        now = time.time()

        # Refill tokens based on time elapsed

        time_passed = now - self._last_refill[provider]

        tokens_to_add = int(time_passed * config.max_requests / config.time_window)

        if tokens_to_add > 0:

            self._tokens[provider] = min(
                config.max_requests,
                self._tokens[provider] + tokens_to_add
            )

            self._last_refill[provider] = now

        # Check if we have tokens available

        if self._tokens[provider] > 0:

            self._tokens[provider] -= 1

            self._request_times[provider].append(now)

        # Clean old request times

        cutoff = now - config.time_window

        while (self._request_times[provider] and
               self._request_times[provider][0] < cutoff):
            self._request_times[provider].popleft()

        return True

    return False
```

```
def get_wait_time(self, provider: str) -> float:
    """Get recommended wait time before next request."""
    config = self.provider_configs[provider]
    with self._lock:
        if not self._request_times[provider]:
            return 0.0

        # Calculate time until oldest request expires
        oldest_request = self._request_times[provider][0]
        time_until_token = (oldest_request + config.time_window) - time.time()
        return max(0.0, time_until_token)
```

Core Logic Skeleton Code:

```
from typing import List, Dict, Any, Optional, Callable

from dataclasses import dataclass

import asyncio

import hashlib

import json

import sqlite3

from pathlib import Path

class BatchProcessor:
```

"""Processes evaluation batches with adaptive sizing and error handling."""

```
def __init__(self, max_concurrent: int = 10, initial_batch_size: int = 20):
```

```
    self.max_concurrent = max_concurrent
```

```
    self.initial_batch_size = initial_batch_size
```

```
    self.batch_stats = {} # provider -> {avg_time, error_rate, optimal_size}
```

```
async def process_batch(self, test_cases: List[TestCase],
```

```
                      processor_func: Callable) -> List[EvaluationResult]:
```

"""Process a batch of test cases with the given processor function.

Args:

```
    test_cases: List of test cases to evaluate
```

```
    processor_func: Async function that processes individual test cases
```

Returns:

```
    List of evaluation results, same order as input test cases
```

"""

```
# TODO 1: Determine optimal batch size based on provider performance history
```

```
# TODO 2: Create semaphore to limit concurrent operations within batch
```

```
# TODO 3: Process test cases concurrently using asyncio.gather()
```

```

# TODO 4: Handle individual test case failures without failing entire batch

# TODO 5: Update batch statistics for future size optimization

# TODO 6: Return results in same order as input (important for progress tracking)

# Hint: Use asyncio.Semaphore(self.max_concurrent) to control concurrency

pass


def _calculate_optimal_batch_size(self, provider: str) -> int:

    """Calculate optimal batch size based on provider performance stats."""

    # TODO 1: Retrieve historical performance stats for provider

    # TODO 2: If error rate > 10%, reduce batch size by 50%

    # TODO 3: If avg completion time > 30 seconds, reduce batch size by 25%

    # TODO 4: If performance is good, gradually increase batch size up to max

    # TODO 5: Ensure batch size stays within bounds [5, 100]

    # Hint: Use exponential moving average for performance metrics

    pass


class PromptCache:

    """Content-addressable cache for LLM responses with TTL and normalization."""

    def __init__(self, cache_dir: Path, default_ttl: int = 30 * 24 * 3600):

        self.cache_dir = cache_dir

        self.default_ttl = default_ttl

        self.db_path = cache_dir / "prompt_cache.db"

        self._init_database()


    def get_cached_response(self, prompt: str, provider_config: LLMPProviderConfig) -> Optional[str]:

        """Retrieve cached response for normalized prompt and provider config."""

        # TODO 1: Normalize prompt text (strip whitespace, normalize line endings)

        # TODO 2: Serialize provider config to canonical JSON (sorted keys)

        # TODO 3: Generate cache key as SHA256(normalized_prompt + config_json)

```

```

# TODO 4: Query SQLite database for cache entry with key

# TODO 5: Check if entry is within TTL, return None if expired

# TODO 6: Return cached response if valid, None otherwise

# Hint: Use hashlib.sha256() and json.dumps(sort_keys=True)

pass


def store_response(self, prompt: str, provider_config: LLMPProviderConfig,
                   response: str, ttl: Optional[int] = None) -> None:
    """Store LLM response in cache with TTL."""

    # TODO 1: Generate same cache key as get_cached_response()

    # TODO 2: Calculate expiration timestamp (current_time + ttl)

    # TODO 3: Insert or replace cache entry in SQLite database

    # TODO 4: Handle database lock errors with retry logic

    # TODO 5: Optionally trigger background cleanup of expired entries

    # Hint: Use "INSERT OR REPLACE" SQL statement for atomic updates

    pass


class CheckpointManager:

    """Manages hierarchical checkpoints with WAL for crash recovery."""

    def __init__(self, checkpoint_dir: Path):
        self.checkpoint_dir = checkpoint_dir

        self.wal_path = checkpoint_dir / "evaluation.wal"

        self.last_full_snapshot = 0 # sequence number


    def write_incremental_checkpoint(self, evaluation_results: List[EvaluationResult]) -> None:
        """Write incremental checkpoint with just the new results."""

        # TODO 1: Generate sequence number (monotonically increasing)

        # TODO 2: Create checkpoint record with sequence, timestamp, results

        # TODO 3: Serialize checkpoint record to JSON

```

```

# TODO 4: Append to WAL file with atomic write (write + fsync)

# TODO 5: Update internal state tracking (last checkpoint sequence)

# Hint: Use os.fsync() to ensure data hits disk before returning

pass


def write_full_snapshot(self, evaluation_run: EvaluationRun) -> None:

    """Write complete evaluation state snapshot."""

    # TODO 1: Generate snapshot filename with timestamp and sequence number

    # TODO 2: Serialize complete EvaluationRun state to JSON

    # TODO 3: Write snapshot to temporary file first

    # TODO 4: Atomically rename temp file to final name (atomic on POSIX)

    # TODO 5: Update last_full_snapshot sequence number

    # TODO 6: Optionally clean up old snapshots and WAL entries

    # Hint: Use tempfile.NamedTemporaryFile() in same directory as target

    pass


def recover_evaluation_state(self) -> Optional[EvaluationRun]:

    """Recover evaluation state from most recent checkpoint + WAL replay."""

    # TODO 1: Find most recent full snapshot file in checkpoint directory

    # TODO 2: Load snapshot and deserialize to EvaluationRun object

    # TODO 3: Scan WAL for entries newer than snapshot sequence number

    # TODO 4: Replay WAL entries in order to update evaluation state

    # TODO 5: Validate recovered state for consistency

    # TODO 6: Return recovered EvaluationRun or None if no valid checkpoint

    # Hint: Handle corrupted files gracefully, try previous snapshots

    pass


class ProgressTracker:

    """Tracks evaluation progress with statistical time estimation."""

```

```

def __init__(self):
    self.start_time: Optional[float] = None
    self.completed_count = 0
    self.total_count = 0
    self.complexity_history = [] # (complexity_features, actual_time) tuples

def update_progress(self, completed_results: List[EvaluationResult]) -> None:
    """Update progress tracking with newly completed results."""
    # TODO 1: Update completed_count with number of new results
    # TODO 2: Extract complexity features and actual times from results
    # TODO 3: Add (features, time) tuples to complexity_history
    # TODO 4: Limit history size to prevent unbounded memory growth
    # TODO 5: Trigger progress display update if enough new completions
    # Hint: Complexity features = [prompt_tokens, expected_tokens, metric_count]
    pass

def estimate_time_remaining(self, pending_cases: List[TestCase]) -> tuple[float, float]:
    """Estimate time remaining with confidence interval."""
    # TODO 1: Extract complexity features for each pending test case
    # TODO 2: Find k-nearest neighbors in complexity_history for each case
    # TODO 3: Estimate time per case as weighted average of neighbors
    # TODO 4: Sum estimated times for all pending cases
    # TODO 5: Calculate confidence interval based on variance in estimates
    # TODO 6: Return (estimated_seconds, confidence_interval_width)
    # Hint: Use scikit-learn's NearestNeighbors if available, else simple distance
    pass

```

Milestone Checkpoints:

After implementing the execution engine:

- Run `python -m pytest tests/test_execution_engine.py -v`

- Create a test dataset with 100 test cases and run evaluation
- Verify batch processing adapts size based on simulated API latency
- Check that rate limiting prevents API overload during high concurrency

After implementing caching:

- Run evaluation twice on same dataset, verify second run is much faster
- Check cache hit rates are reported correctly in progress output
- Test cache invalidation by changing provider configuration
- Verify cache storage doesn't grow unbounded over multiple runs

After implementing checkpointing:

- Start large evaluation, kill process partway through with Ctrl+C
- Restart evaluation and verify it resumes from checkpoint
- Check WAL file contains incremental progress records
- Test recovery from corrupted checkpoint (should fall back gracefully)

After implementing progress tracking:

- Run evaluation and verify time estimates become more accurate over time
- Test with mixed complexity test cases (short vs long prompts)
- Check confidence intervals reflect estimate uncertainty
- Verify progress display updates smoothly without flickering

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
Evaluation hangs without progress	Rate limiter too restrictive or API provider down	Check rate limiter wait times, test API directly	Adjust rate limits or switch providers
Memory usage grows unboundedly	Cache not expiring old entries or checkpoint history growing	Monitor cache size, check WAL cleanup	Implement TTL cleanup and checkpoint pruning
Time estimates wildly inaccurate	Insufficient complexity modeling or outlier test cases	Analyze complexity_history for patterns	Add more complexity features or outlier detection
Frequent checkpoint corruption	System shutdown during writes or disk full	Check disk space, examine checkpoint write timing	Use atomic writes and validate disk space
Poor cache hit rates	Prompt normalization too strict or provider configs changing	Compare cache keys for similar prompts	Adjust normalization rules or cache invalidation

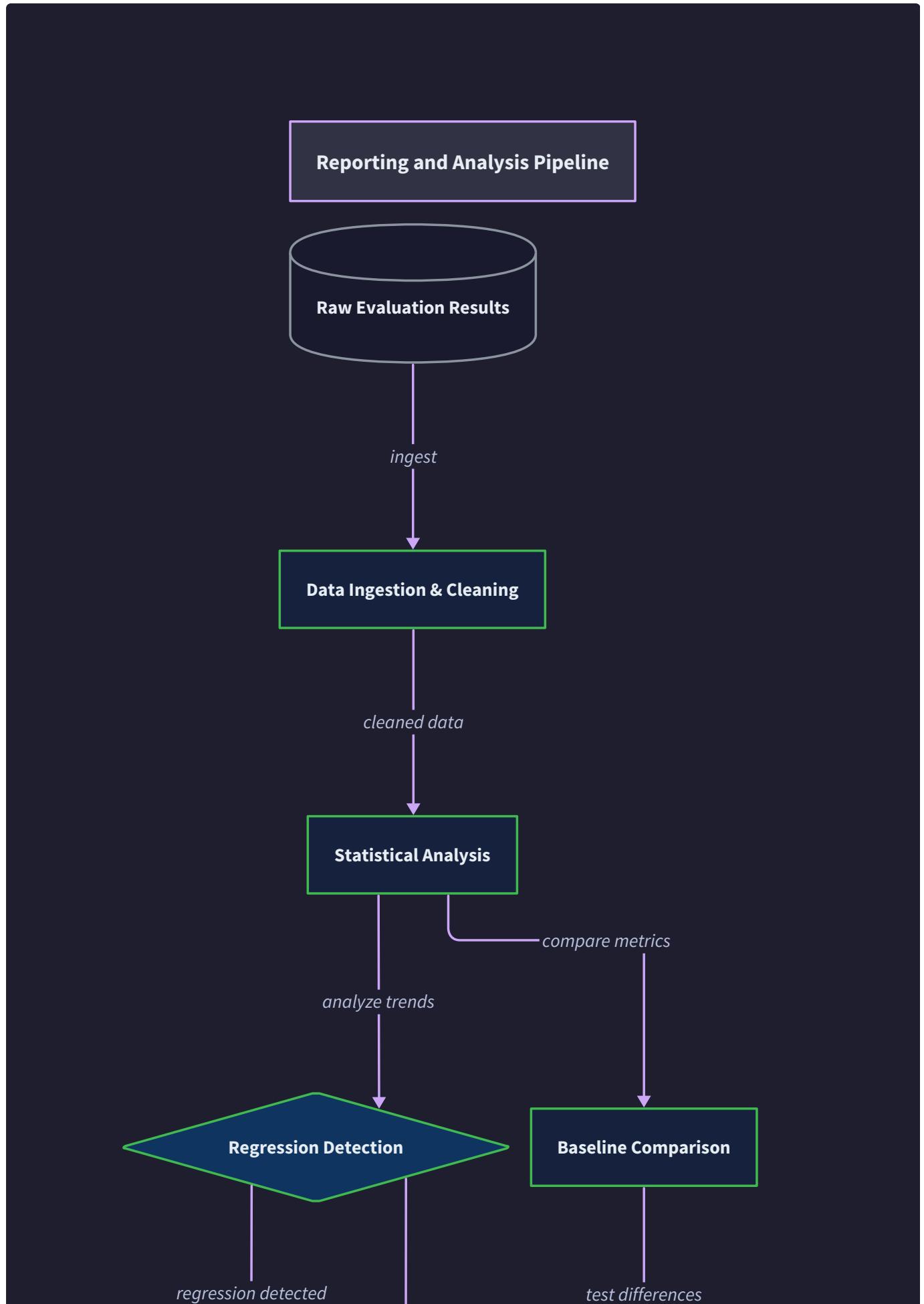
Reporting and Analysis

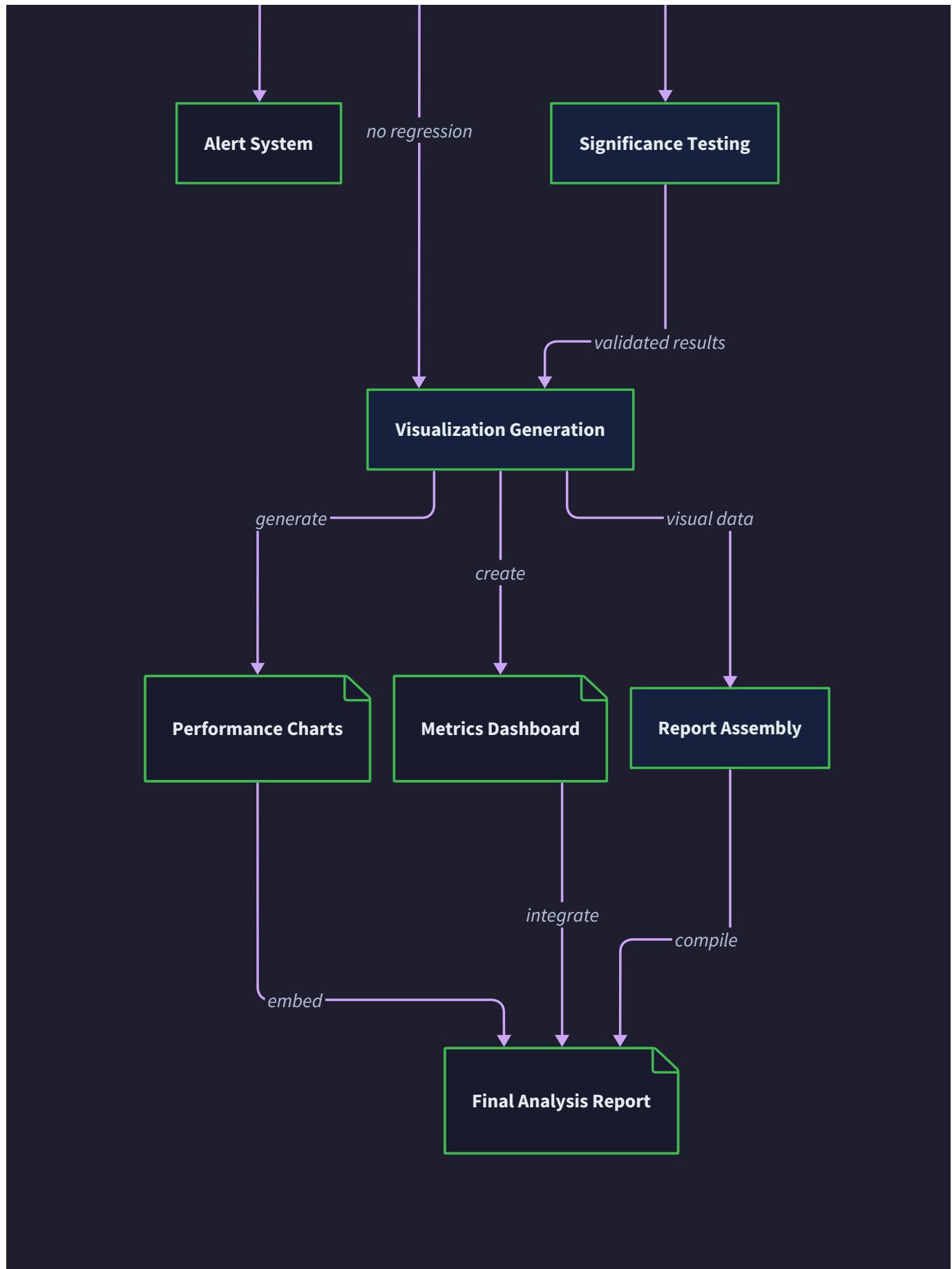
Milestone(s): Milestone 4 (Reporting & Analysis) - This section covers the complete statistical analysis engine that transforms raw evaluation results into actionable insights through score aggregation, regression detection, failure analysis, and comprehensive report generation

The reporting and analysis system represents the culmination of the evaluation framework — the component that transforms raw evaluation data into actionable insights for model improvement. Think of it as a **data scientist's workbench** built into the evaluation pipeline: it doesn't just show you numbers, but automatically surfaces patterns, detects problems, and generates hypotheses about what's working and what needs attention.

Unlike traditional software testing where pass/fail is binary, LLM evaluation produces rich, multidimensional data that requires sophisticated analysis to be meaningful. A single evaluation run might generate thousands of scored responses across dozens of metrics and categories. The reporting system must synthesize this complexity into clear, actionable insights while preserving the statistical rigor needed for confident decision-making.

The analysis engine operates on four levels of insight generation: **descriptive statistics** that summarize current performance, **comparative analysis** that detects changes over time, **diagnostic analysis** that identifies failure patterns, and **prescriptive insights** that suggest improvement directions. Each level builds on the previous one, creating a comprehensive understanding of model performance.





Decision: Automated Statistical Analysis vs Manual Report Building

- **Context:** Organizations need to process evaluation results at scale without requiring statistical expertise from every user
- **Options Considered:** Manual analysis with raw data exports, template-based reporting, fully automated statistical analysis

- **Decision:** Fully automated statistical analysis with configurable thresholds and customizable report templates
- **Rationale:** Manual analysis doesn't scale and requires statistical expertise. Template-based reporting lacks the sophistication needed for LLM evaluation complexity. Automated analysis ensures consistent methodology while remaining accessible to non-statisticians.
- **Consequences:** Enables rapid insight generation and reduces time-to-decision. Requires sophisticated statistical algorithms and careful handling of edge cases like small sample sizes.

Score Aggregation

Score aggregation transforms individual metric results into meaningful statistical summaries that reveal performance patterns across different dimensions of the evaluation dataset. Think of it as creating a **multidimensional performance map** — instead of looking at thousands of individual scores, you get clear views of performance by category, difficulty level, and other meaningful groupings.

The aggregation system must handle the inherent complexity of LLM evaluation data: different metrics have different scales and interpretations, test cases belong to multiple overlapping categories, and statistical significance varies dramatically with sample size. The goal is to provide both high-level summaries for quick decision-making and detailed breakdowns for deep analysis.

The aggregation engine processes evaluation results through multiple statistical lenses simultaneously. **Overall performance metrics** provide system-wide health indicators, while **category-based breakdowns** reveal strengths and weaknesses in specific areas. **Tag-based analysis** enables fine-grained performance investigation, and **difficulty-stratified results** help understand model capabilities across different challenge levels.

Decision: Multi-Level Aggregation vs Flat Summarization

- **Context:** Evaluation results need to be interpretable at different levels of granularity for different stakeholders
- **Options Considered:** Single overall score, hierarchical breakdown by tags/categories, configurable aggregation dimensions
- **Decision:** Multi-level hierarchical aggregation with automatic statistical significance testing
- **Rationale:** Different stakeholders need different views — executives want overall trends, engineers need category-specific insights, researchers need tag-level details. Statistical significance testing prevents over-interpretation of small sample results.
- **Consequences:** Provides comprehensive insights but requires careful statistical methodology and clear presentation of confidence levels.

The **statistical foundation** of score aggregation rests on robust descriptive statistics that account for the unique characteristics of evaluation data. Each aggregation level computes central tendency measures (mean, median), variability measures (standard deviation, interquartile range), and distributional characteristics (skewness, outlier detection). The system automatically adjusts statistical methods based on sample size and distribution properties.

Statistical Measure	Purpose	Calculation Method	Interpretation Guidelines
Mean Score	Central tendency	Arithmetic mean with outlier handling	Primary performance indicator, sensitive to extreme values
Median Score	Robust central tendency	50th percentile	Less affected by outliers, better for skewed distributions
Standard Deviation	Variability measure	Population standard deviation	Indicates consistency, higher values suggest uneven performance
Confidence Interval	Uncertainty bounds	Bootstrap or t-distribution based	95% CI for mean score, wider intervals indicate less certainty
Sample Size	Statistical power	Count of valid results	Minimum 30 for reliable statistics, flag small samples
Percentile Breakdown	Distribution shape	10th, 25th, 75th, 90th percentiles	Reveals distribution characteristics and outlier patterns

Category-based aggregation organizes results according to the natural structure of the evaluation dataset. The system automatically discovers categories from test case tags and metadata, then computes comprehensive statistics for each category. This enables identification of systematic strengths and weaknesses — for example, consistently high performance on factual questions but poor performance on creative tasks.

The aggregation algorithm handles **overlapping categories** intelligently. Since test cases can belong to multiple categories (e.g., both "medical" and "factual"), the system avoids double-counting while still providing complete category-specific insights. Each test case contributes to all applicable category statistics, with clear tracking of overlap patterns for transparency.

Aggregation Level	Scope	Statistical Measures	Use Cases
Overall	All test cases	Mean, median, std dev, CI, percentiles	System health monitoring, executive dashboards
Category	Test cases by category tags	Per-category stats, category comparison	Domain-specific performance analysis
Tag	Test cases by individual tags	Tag-specific performance, tag correlation	Fine-grained debugging, feature analysis
Difficulty	Test cases by difficulty level	Performance vs difficulty curve	Capability assessment, training prioritization
Temporal	Results over time windows	Trend analysis, change detection	Performance monitoring, regression tracking

Score normalization and weighting ensures meaningful comparisons across different metrics and categories. The system applies configurable normalization schemes to handle metrics with different scales and distributions. **Equal**

weighting treats all metrics as equally important, while **custom weighting** allows domain experts to emphasize critical evaluation dimensions.

The **temporal aggregation** component tracks performance changes over time by computing rolling statistics and trend indicators. This enables detection of gradual performance changes that might be missed in single-evaluation snapshots. The system maintains statistical baselines and computes change significance to distinguish real trends from random fluctuation.

Key Insight: Statistical Significance in Practice Small sample sizes plague many LLM evaluations, making statistical significance testing crucial. The aggregation system automatically flags results with insufficient statistical power and adjusts confidence intervals accordingly. This prevents over-interpretation of results based on too little data.

Aggregation Configuration allows customization of the statistical analysis to match organizational needs and domain requirements. Users can specify minimum sample sizes for reliable statistics, choose statistical methods appropriate for their data characteristics, and configure alerting thresholds for performance changes.

Configuration Parameter	Purpose	Default Value	Impact
<code>min_sample_size</code>	Minimum cases for reliable statistics	30	Controls statistical significance flags
<code>confidence_level</code>	Confidence interval width	0.95	Affects uncertainty bound calculations
<code>outlier_threshold</code>	Standard deviations for outlier detection	2.5	Balances outlier sensitivity vs false positives
<code>normalization_method</code>	Score normalization approach	"min_max"	Determines cross-metric comparison validity
<code>weighting_scheme</code>	Metric importance weighting	"equal"	Controls relative metric influence
<code>temporal_window</code>	Time window for trend analysis	30 days	Balances trend sensitivity vs noise

Common Pitfalls in Score Aggregation

⚠ **Pitfall: Simpson's Paradox in Category Aggregation** When aggregating scores across categories, the overall average can show improvement while all individual categories show decline (or vice versa) due to changing category composition. The aggregation system addresses this by tracking category distribution changes and flagging potential Simpson's paradox cases with detailed composition analysis.

⚠ **Pitfall: Statistical Significance with Multiple Comparisons** When computing statistics for dozens of categories simultaneously, random chance will produce apparently significant results even when no real differences exist. The system applies multiple comparison correction (Bonferroni or FDR) when flagging category differences and clearly indicates adjusted vs unadjusted significance levels.

⚠ **Pitfall: Metric Scale Incompatibility** Aggregating metrics with fundamentally different scales (0-1 probabilities vs 0-100 BLEU scores) without proper normalization produces meaningless results. The system enforces metric metadata requirements that specify scale and distribution characteristics, then applies appropriate normalization before aggregation.

Regression Detection

Regression detection serves as the evaluation framework's **early warning system**, automatically identifying when model performance has degraded compared to established baselines. Think of it as a **continuous integration system for AI quality** — just as CI catches code regressions before they reach production, this system catches performance regressions before they impact users.

The challenge of regression detection in LLM evaluation goes far beyond simple metric comparison. Performance can degrade in subtle ways that affect only specific categories or edge cases, changes can be gradual rather than sudden, and statistical noise can mask real degradation or create false alarms. The detection system must distinguish meaningful performance changes from random variation while being sensitive enough to catch important regressions early.

Baseline management forms the foundation of effective regression detection. The system maintains multiple types of baselines: **golden baselines** representing the best-ever performance, **production baselines** reflecting current deployed model performance, and **development baselines** tracking progress during model iteration. Each baseline includes not just aggregate scores but complete distributional characteristics needed for rigorous statistical comparison.

Decision: Statistical Testing vs Threshold-Based Detection

- **Context:** Need to distinguish real performance changes from random variation in noisy evaluation metrics
- **Options Considered:** Simple threshold comparison, statistical hypothesis testing, machine learning anomaly detection
- **Decision:** Statistical hypothesis testing with configurable significance levels and effect size thresholds
- **Rationale:** Thresholds alone can't account for natural variation in evaluation metrics. Statistical testing provides principled uncertainty quantification. ML anomaly detection adds complexity without clear benefits for this use case.
- **Consequences:** Enables confident regression detection with quantified uncertainty. Requires careful statistical methodology and clear interpretation guidelines.

The **statistical testing framework** applies appropriate hypothesis tests based on data characteristics and comparison requirements. For normally distributed metrics, the system uses **t-tests** or **Welch's t-test** when variances differ. For non-normal distributions, it applies **Mann-Whitney U tests** or other non-parametric alternatives. The framework automatically selects appropriate tests based on data distribution analysis and sample size considerations.

Comparison Type	Statistical Test	Requirements	Interpretation
Single metric, normal distribution	Two-sample t-test	$n > 30$, normality confirmed	p-value $< \alpha$ indicates significant change
Single metric, non-normal	Mann-Whitney U test	No distribution assumptions	Rank-based comparison, robust to outliers
Multiple metrics	Multivariate Hotelling T^2	Correlated metrics	Tests for overall performance vector change
Time series	Changepoint detection	Sequential data	Identifies when performance regime shifted
Category-specific	Stratified analysis	Sufficient per-category samples	Detects localized performance changes

Effect size estimation ensures that statistically significant changes are also practically meaningful. A large evaluation dataset might detect tiny performance differences that are statistically significant but practically irrelevant. The system computes **Cohen's d** for metric differences and flags only changes that exceed both statistical significance and practical significance thresholds.

The **multi-dimensional regression analysis** simultaneously monitors performance across all evaluation dimensions to detect subtle degradations that might be missed by overall metrics. This includes category-specific regressions, tag-level changes, and difficulty-stratified analysis. The system applies multiple comparison corrections to control false discovery rates when testing many dimensions simultaneously.

Regression Detection Dimension	Analysis Method	Alert Criteria	Action Recommended
Overall Performance	Global metric comparison	$p < 0.05$, effect size > 0.2	Immediate investigation
Category-Specific	Per-category statistical tests	FDR-corrected $p < 0.05$	Domain expert review
Tag-Level	Fine-grained tag analysis	Multiple tags affected	Pattern analysis
Difficulty-Stratified	Performance vs difficulty	Slope change $>$ threshold	Capability assessment
Temporal Pattern	Trend analysis	Significant negative slope	Long-term monitoring
Distribution Shape	Kolmogorov-Smirnov test	Distribution shift detected	Data quality investigation

Changepoint detection algorithms identify when performance characteristics fundamentally shift, even if the change is gradual. The system applies **CUSUM (Cumulative Sum)** control charts and **Bayesian changepoint detection** to identify the specific evaluation run when performance regime changed. This enables precise timing of regression onset and correlation with model changes or data updates.

The **adaptive baseline system** automatically updates baseline expectations as models improve, preventing false regression alerts when the current model significantly outperforms historical baselines. However, it maintains **golden standard baselines** that represent the best-ever performance to detect any retreat from peak capabilities.

Key Insight: The Baseline Staleness Problem Baselines become less relevant over time as evaluation datasets evolve and model capabilities advance. The system addresses this by maintaining multiple baseline types with different update policies: recent baselines for short-term regression detection and stable baselines for long-term capability tracking.

Regression alert configuration allows teams to customize detection sensitivity based on their risk tolerance and evaluation methodology. High-stakes production systems might require very sensitive detection with low significance thresholds, while research environments might prefer higher thresholds to reduce alert fatigue.

Configuration Parameter	Purpose	Typical Values	Trade-off
<code>significance_threshold</code>	p-value threshold for statistical significance	0.01, 0.05, 0.10	Lower = fewer false positives, more missed regressions
<code>effect_size_threshold</code>	Minimum effect size for practical significance	0.1, 0.2, 0.3	Higher = fewer trivial alerts, might miss subtle issues
<code>baseline_update_policy</code>	How frequently baselines are updated	"manual", "monthly", "best_ever"	More frequent = fewer false alerts, less sensitivity
<code>multiple_comparison_correction</code>	Method for adjusting p-values	"bonferroni", "fdr", "none"	Stricter = fewer false discoveries, more missed detections
<code>temporal_sensitivity</code>	Lookback period for trend detection	7, 14, 30 days	Longer = smoother trends, slower detection

Failure Analysis

Failure analysis transforms individual evaluation failures into systematic insights about model limitations and improvement opportunities. Think of it as **forensic analysis for AI systems** — instead of just knowing that some test cases failed, you understand the underlying patterns, root causes, and specific areas where the model struggles.

The challenge lies in the sheer volume and diversity of failure modes in LLM evaluation. A single evaluation run might produce hundreds of incorrect responses across dozens of categories and failure types. Manual analysis of this scale is impractical, yet automated analysis must preserve the nuanced understanding that human experts bring to failure interpretation.

The **failure clustering algorithm** groups similar failures to reveal systematic patterns rather than treating each failure as an isolated incident. The system applies **semantic clustering** using embedding representations of failed responses, **error type classification** based on rubric violations, and **root cause analysis** that traces failures back to likely model limitations.

Decision: Semantic Clustering vs Rule-Based Classification

- **Context:** Need to group diverse failure modes into meaningful categories for analysis
- **Options Considered:** Manual categorization, rule-based error classification, semantic similarity clustering
- **Decision:** Hybrid approach combining semantic clustering with rule-based classification and human validation
- **Rationale:** Pure rule-based systems miss novel failure modes. Pure clustering produces clusters that aren't interpretable. Hybrid approach provides both comprehensiveness and interpretability.
- **Consequences:** Enables discovery of unexpected failure patterns while maintaining interpretable categories. Requires more sophisticated implementation and occasional human calibration.

Semantic failure clustering uses embedding representations to group failures with similar semantic characteristics, even when surface-level responses differ significantly. For example, all instances where the model provides factually incorrect medical information might cluster together regardless of the specific medical domain or question format.

The clustering algorithm applies **hierarchical clustering** with **cosine similarity** on response embeddings, automatically determining the optimal number of clusters using **silhouette analysis** and **gap statistic** methods. Each cluster receives automatic labeling based on common keywords and patterns in the clustered failures.

Clustering Dimension	Method	Purpose	Output
Semantic Content	Embedding cosine similarity	Groups failures by meaning	"Medical misinformation", "Math calculation errors"
Error Type	Rule-based classification	Groups by error category	"Hallucination", "Refusal", "Format violation"
Input Characteristics	Feature-based clustering	Groups by input properties	"Long prompts", "Multi-step reasoning", "Domain-specific"
Response Quality	Quality metric clustering	Groups by degradation type	"Partial answers", "Completely wrong", "Right fact, wrong context"

Root cause analysis attempts to trace failure patterns back to likely model limitations or training data issues. The system analyzes failure clusters in context of input characteristics, expected outputs, and model training information to generate **actionable hypotheses** about improvement strategies.

The analysis engine maintains a **failure taxonomy** that categorizes common LLM failure modes and maps them to potential remediation approaches. This enables not just failure description but specific recommendations for addressing systematic issues.

Failure Category	Characteristics	Likely Root Cause	Suggested Remediation
Factual Hallucination	Confident incorrect facts	Training data gaps or retrieval issues	Knowledge base enhancement, retrieval augmentation
Reasoning Chain Breaks	Correct start, wrong conclusion	Multi-step reasoning limitations	Chain-of-thought training, reasoning verification
Context Confusion	Right info, wrong context	Context window or attention issues	Context management training, prompt engineering
Format Violations	Correct content, wrong format	Instruction following weaknesses	Format-specific fine-tuning, template training
Refusal Over-caution	Unnecessary refusal to answer	Safety training over-generalization	Balanced safety training, edge case examples
Inconsistent Responses	Different answers to same question	Training data inconsistencies	Data deduplication, consistency training

Failure impact analysis quantifies the business or user impact of different failure types to prioritize improvement efforts. The system weighs failure frequency against severity and user-facing consequences to generate **improvement priority rankings**.

The **comparative failure analysis** compares failure patterns across different model versions, configurations, or time periods to identify trends and measure improvement progress. This enables tracking whether specific interventions successfully addressed targeted failure modes.

Analysis Dimension	Comparison Method	Insight Type	Application
Temporal Trends	Failure rate over time	Improving vs degrading categories	Progress tracking
Model Comparison	Side-by-side failure analysis	Relative strengths/weaknesses	Model selection
Configuration Impact	A/B failure pattern comparison	Configuration effectiveness	Hyperparameter optimization
Dataset Evolution	Failure rates on dataset versions	Data quality impact	Dataset improvement

Interactive failure exploration enables human experts to dive deep into specific failure clusters, examine representative examples, and provide human insights that enhance automated analysis. The system presents failure clusters with representative examples, similarity explanations, and hypothesized root causes for expert validation and refinement.

Key Insight: The Long Tail of Failure Modes LLM failures follow a power law distribution — a few failure modes account for most failures, but the long tail of rare failure modes can be critically important for specific use cases. The failure analysis system balances attention between high-frequency failures and potentially high-impact rare failures.

Report Generation

Report generation synthesizes all analysis outputs into comprehensive, actionable documents that serve different stakeholder needs within the organization. Think of it as creating a **personalized data story** for each audience — the same underlying evaluation data becomes an executive summary for leadership, a technical deep-dive for engineers, and a research report for model developers.

The report generation system must handle multiple output formats (HTML, PDF, interactive dashboards), different levels of technical detail, and various time horizons (single evaluation reports, longitudinal studies, comparative analyses). Most critically, reports must be **actionable** — they should not just present information but guide specific decisions and improvement actions.

Multi-audience report generation creates tailored views of the same evaluation results. **Executive summaries** focus on high-level trends, business impact, and strategic recommendations. **Engineering reports** emphasize actionable failure patterns, performance breakdowns, and implementation recommendations. **Research reports** provide detailed statistical analyses, methodological notes, and detailed failure case studies.

Decision: Template-Based vs Dynamic Report Generation

- **Context:** Need to generate professional reports quickly while maintaining flexibility for different use cases
- **Options Considered:** Fixed report templates, completely custom generation, hybrid template system with dynamic sections
- **Decision:** Hybrid template system with customizable sections and dynamic content generation
- **Rationale:** Fixed templates are too rigid for diverse use cases. Completely custom generation requires too much manual work. Hybrid approach provides professional formatting with content flexibility.
- **Consequences:** Enables rapid report generation with professional appearance and customizable content. Requires template maintenance and dynamic content generation logic.

The **executive report format** distills complex evaluation results into business-focused insights with clear recommendations. These reports emphasize trends over absolute numbers, focus on user impact over technical metrics, and provide specific next steps rather than detailed analysis.

Executive Report Section	Content Focus	Visualization Type	Key Messages
Performance Summary	Overall trends, key changes	Trend lines, scorecards	"Is performance improving?"
Business Impact	User-facing implications	Impact matrices, priority charts	"What matters most to users?"
Risk Assessment	Potential issues, degradation areas	Risk heat maps, alert summaries	"What should we worry about?"
Recommendations	Specific next steps	Priority lists, roadmap views	"What should we do next?"
Resource Requirements	Investment needed for improvements	Resource allocation charts	"What will it cost to improve?"

Technical engineering reports provide the detailed analysis that development teams need for specific improvement actions. These reports emphasize failure patterns, performance bottlenecks, and implementation-specific recommendations with sufficient detail for immediate action.

The technical report format includes **code-level recommendations** where appropriate, **specific test cases** that illustrate issues, and **quantitative improvement targets** based on observed performance gaps.

Technical Report Section	Content Focus	Detail Level	Actionability
Performance Analysis	Metric breakdowns, statistical significance	Detailed statistics, confidence intervals	Specific optimization targets
Failure Deep-Dive	Clustered failures, root cause analysis	Individual examples, pattern analysis	Specific fixes to implement
Regression Analysis	Performance changes, degradation areas	Statistical tests, effect sizes	Rollback or improvement decisions
Category Performance	Domain-specific insights	Per-category detailed analysis	Domain-specific improvement strategies
Implementation Guide	Specific next steps, code recommendations	Step-by-step instructions	Immediately actionable tasks

Interactive report features enable stakeholders to explore results beyond the static report content. Web-based reports include **drill-down capabilities** that allow exploration from high-level summaries to individual test case results, **filter controls** for focusing on specific categories or time periods, and **comparison tools** for side-by-side analysis of different evaluation runs.

The **visualization system** automatically selects appropriate chart types based on data characteristics and stakeholder needs. **Distribution plots** reveal performance patterns, **trend lines** show changes over time, **heat maps** highlight category-specific issues, and **scatter plots** reveal correlations between different metrics.

Visualization Type	Data Type	Use Case	Interactive Features
Score Distribution	Metric scores	Performance spread analysis	Histogram bins, outlier highlighting
Trend Analysis	Time series	Performance over time	Date range selection, metric filtering
Category Heatmap	Category × Metric	Systematic strength/weakness identification	Category drill-down, metric selection
Regression Scatter	Current vs baseline	Change magnitude visualization	Point details, regression line toggle
Failure Sunburst	Hierarchical failure categories	Failure pattern exploration	Category expansion, sample selection
Comparison Matrix	Multiple evaluation runs	Side-by-side performance comparison	Run selection, metric highlighting

Report customization enables organizations to adapt report formats to their specific needs, branding requirements, and stakeholder preferences. The system provides **template customization** for consistent organizational branding, **section configuration** for including only relevant analyses, and **metric selection** for focusing on business-critical measurements.

The **automated report scheduling** system generates and distributes reports on configurable schedules, ensuring stakeholders receive timely updates without manual intervention. Reports can be triggered by evaluation completion, scheduled at regular intervals, or generated on-demand for specific analysis needs.

Key Insight: The Actionability Principle Every element in a report should either inform a decision or suggest a specific action. If a chart, table, or analysis doesn't lead to actionable insights, it creates noise rather than value. The report generation system applies this filter ruthlessly to ensure maximum utility.

Report validation and quality control ensures that generated reports are accurate, complete, and professionally presented. The system validates data consistency across report sections, checks for sufficient statistical power before making claims, and applies formatting consistency rules for professional appearance.

Common Pitfalls in Reporting and Analysis

⚠ Pitfall: Over-Interpretation of Noisy Data LLM evaluation metrics are inherently noisy, and small changes can appear significant when viewed in isolation. The reporting system addresses this by always presenting confidence intervals, requiring minimum sample sizes for strong claims, and prominently displaying uncertainty indicators in all analyses.

⚠ Pitfall: Chart Junk and Information Overload Complex evaluation results can lead to cluttered visualizations that obscure rather than illuminate insights. The system applies strict information design principles: every visual element must serve a specific analytical purpose, color is used strategically to highlight important information, and complexity is progressively disclosed through interactive features.

⚠ Pitfall: Static Analysis of Dynamic Systems LLM models and evaluation datasets both evolve over time, making point-in-time reports potentially misleading. The system addresses this by incorporating temporal context in all analyses,

highlighting when historical comparisons may not be valid due to dataset changes, and providing guidance on when baseline updates are needed.

Implementation Guidance

The reporting and analysis system requires sophisticated statistical analysis capabilities combined with flexible report generation. The implementation balances statistical rigor with practical usability for different stakeholder audiences.

Technology Recommendations

Component	Simple Option	Advanced Option
Statistical Analysis	scipy.stats + numpy for basic statistics	statsmodels + scipy for advanced statistical testing
Visualization	matplotlib + seaborn for static charts	plotly + dash for interactive visualizations
Report Generation	jinja2 templates + weasyprint for PDF	full web framework with dynamic content generation
Data Processing	pandas for tabular data analysis	polars for high-performance data processing
Clustering	scikit-learn for basic clustering	custom embedding-based clustering with sentence-transformers

File Structure

```
reporting/
    __init__.py
aggregation/
    __init__.py
    score_aggregator.py      ← Statistical aggregation engine
    baseline_manager.py      ← Baseline management and comparison
    significance_testing.py  ← Statistical hypothesis testing
regression/
    __init__.py
    detector.py              ← Regression detection algorithms
    changepoint.py           ← Changepoint detection methods
    alerts.py                ← Alert generation and configuration
failure_analysis/
    __init__.py
    clustering.py            ← Failure clustering algorithms
    root_cause.py            ← Root cause analysis engine
    taxonomy.py              ← Failure classification system
report_generation/
    __init__.py
    template_engine.py       ← Report template management
    visualizations.py        ← Chart and graph generation
    export_formats.py        ← PDF, HTML, dashboard export
templates/
    executive_summary.html  ← Executive report template
    technical_report.html   ← Engineering report template
    research_analysis.html  ← Detailed research template
tests/
    test_aggregation.py
    test_regression.py
    test_failure_analysis.py
    test_report_generation.py
```

Score Aggregation Implementation

```
from dataclasses import dataclass  
  
from typing import Dict, List, Optional, Tuple  
  
import numpy as np  
  
from scipy import stats  
  
from sklearn.preprocessing import MinMaxScaler  
  
import pandas as pd  
  
  
@dataclass  
  
class AggregationConfig:  
  
    """Configuration for statistical aggregation analysis."""  
  
    min_sample_size: int = 30  
  
    confidence_level: float = 0.95  
  
    outlier_threshold: float = 2.5  
  
    normalization_method: str = "min_max" # "min_max", "z_score", "robust"  
  
    weighting_scheme: str = "equal" # "equal", "custom", "importance"  
  
    temporal_window_days: int = 30  
  
  
@dataclass  
  
class AggregationResult:  
  
    """Results from statistical aggregation analysis."""  
  
    mean_score: float  
  
    median_score: float  
  
    std_deviation: float  
  
    confidence_interval: Tuple[float, float]  
  
    sample_size: int  
  
    percentiles: Dict[int, float]  
  
    outlier_count: int  
  
    distribution_type: str  
  
    statistical_power: float
```

```
class ScoreAggregator:

    """Statistical aggregation engine for evaluation results."""

    def __init__(self, config: AggregationConfig):
        self.config = config
        self.scaler = MinMaxScaler() if config.normalization_method == "min_max" else None

    def compute_overall_statistics(self, evaluation_results: List[EvaluationResult]) ->
        AggregationResult:
        """
        Compute comprehensive statistical summary for all evaluation results.

        Args:
            evaluation_results: List of evaluation results to analyze

        Returns:
            AggregationResult with complete statistical summary
        """
        # TODO 1: Extract overall scores from evaluation results into numpy array
        # TODO 2: Remove invalid scores (NaN, None, out of valid range)
        # TODO 3: Detect and handle outliers based on configured threshold
        # TODO 4: Test for normality using Shapiro-Wilk test (if n < 5000) or Anderson-Darling
        # TODO 5: Compute basic statistics (mean, median, std, percentiles)
        # TODO 6: Calculate appropriate confidence interval based on distribution
        # TODO 7: Estimate statistical power for detecting meaningful effect sizes
        # TODO 8: Package results into AggregationResult object

    def compute_category_breakdown(self, evaluation_results: List[EvaluationResult]) ->
        Dict[str, AggregationResult]:
```

```
"""
```

```
Compute statistical summaries broken down by test case categories.
```

```
Returns:
```

```
Dictionary mapping category names to their statistical summaries
```

```
"""
```

```
# TODO 1: Group evaluation results by test case categories/tags
```

```
# TODO 2: Handle overlapping categories (test cases in multiple categories)
```

```
# TODO 3: For each category, compute statistical summary using compute_overall_statistics
```

```
# TODO 4: Apply multiple comparison correction for significance testing
```

```
# TODO 5: Flag categories with insufficient sample sizes
```

```
# TODO 6: Return category mapping with statistical summaries
```

```
def detect_statistical_significance(self, results_a: List[float], results_b: List[float]) -> Tuple[float, float, str]:
```

```
"""
```

```
Test for statistically significant differences between two result sets.
```

```
Returns:
```

```
Tuple of (p_value, effect_size, test_used)
```

```
"""
```

```
# TODO 1: Check sample sizes and validate input data
```

```
# TODO 2: Test both samples for normality
```

```
# TODO 3: Choose appropriate statistical test (t-test, Mann-Whitney U, etc.)
```

```
# TODO 4: Perform statistical test and compute p-value
```

```
# TODO 5: Calculate effect size (Cohen's d or similar)
```

```
# TODO 6: Return results with test metadata
```

Regression Detection Implementation

```
from dataclasses import dataclass  
  
from datetime import datetime, timedelta  
  
from typing import List, Optional, Tuple  
  
import numpy as np  
  
from scipy import stats  
  
from scipy.signal import find_peaks  
  
import pandas as pd  
  
  
@dataclass  
  
class RegressionAlert:  
  
    """Alert information for detected performance regression."""  
  
    alert_id: str  
  
    detection_time: datetime  
  
    regression_type: str # "overall", "category", "metric"  
  
    affected_components: List[str]  
  
    severity: str # "critical", "major", "minor"  
  
    p_value: float  
  
    effect_size: float  
  
    baseline_performance: float  
  
    current_performance: float  
  
    recommendation: str  
  
  
class RegressionDetector:  
  
    """Automated regression detection system."""  
  
  
    def __init__(self, significance_threshold: float = 0.05,  
                 effect_size_threshold: float = 0.2):  
        self.significance_threshold = significance_threshold  
        self.effect_size_threshold = effect_size_threshold  
        self.baseline_cache = {}  
  
    def detect_regression(self, data: pd.DataFrame) -> Optional[RegressionAlert]:  
        # Implementation of regression detection logic  
        # Returns None if no regression is detected  
        # Returns a RegressionAlert object if a regression is detected  
        pass
```

```

def detect_performance_regression(self,
                                    current_results: List[EvaluationResult],
                                    baseline_results: List[EvaluationResult]) ->
List[RegressionAlert]:
    """
    Detect performance regressions by comparing current results to baseline.

    Returns:
        List of RegressionAlert objects for detected issues
    """
    # TODO 1: Extract and validate score arrays from current and baseline results
    # TODO 2: Perform overall performance comparison using appropriate statistical test
    # TODO 3: Test for significant performance degradation (one-tailed test)
    # TODO 4: If significant, compute effect size and practical significance
    # TODO 5: Perform category-level regression testing with multiple comparison correction
    # TODO 6: Test individual metrics for regression patterns
    # TODO 7: Generate RegressionAlert objects for each detected regression
    # TODO 8: Prioritize alerts by severity (combination of p-value and effect size)

def detect_changepoint(self, time_series_results: List[Tuple[datetime, float]]) ->
Optional[datetime]:
    """
    Detect when performance characteristics fundamentally changed.

    Returns:
        Datetime when changepoint occurred, or None if no significant change
    """
    # TODO 1: Convert time series to numpy arrays for analysis
    # TODO 2: Apply CUSUM algorithm to detect mean shift

```

```
# TODO 3: Use Bayesian changepoint detection for probabilistic analysis

# TODO 4: Validate detected changepoints with statistical significance testing

# TODO 5: Return most likely changepoint datetime or None

def update_baseline(self, new_results: List[EvaluationResult],
                    update_policy: str = "manual") -> bool:
    """
    Update baseline expectations based on new results and update policy.

    Returns:
        True if baseline was updated, False otherwise
    """
    # TODO 1: Validate new results meet minimum quality criteria

    # TODO 2: Apply update policy logic ("manual", "automatic", "best_ever")

    # TODO 3: Compute statistical characteristics of new baseline

    # TODO 4: Store updated baseline with metadata (update time, sample size)

    # TODO 5: Log baseline update for audit trail
```

Failure Analysis Implementation

```
from dataclasses import dataclass  
  
from typing import Dict, List, Optional, Set  
  
import numpy as np  
  
from sklearn.cluster import AgglomerativeClustering  
  
from sklearn.metrics import silhouette_score  
  
from sentence_transformers import SentenceTransformer  
  
import pandas as pd  
  
  
@dataclass  
  
class FailureCluster:  
  
    """Represents a cluster of similar failures."""  
  
    cluster_id: str  
  
    failure_cases: List[str] # Case IDs  
  
    representative_examples: List[str]  
  
    cluster_label: str  
  
    similarity_score: float  
  
    root_cause_hypothesis: str  
  
    suggested_remediation: str  
  
  
class FailureAnalyzer:  
  
    """Comprehensive failure pattern analysis system."""  
  
  
    def __init__(self, embedding_model: str = "all-MiniLM-L6-v2"):  
  
        self.embedding_model = SentenceTransformer(embedding_model)  
  
        self.failure_taxonomy = self._load_failure_taxonomy()  
  
  
    def cluster_failures(self, failed_results: List[EvaluationResult]) -> List[FailureCluster]:  
  
        """  
  
        Cluster similar failures to identify systematic patterns.  
    
```

```

    Returns:
        List of FailureCluster objects representing distinct failure modes
    """
    # TODO 1: Extract failed responses and expected outputs from results
    # TODO 2: Generate embeddings for failed responses using sentence transformer
    # TODO 3: Apply hierarchical clustering with cosine similarity
    # TODO 4: Determine optimal number of clusters using silhouette analysis
    # TODO 5: Generate descriptive labels for each cluster using keyword analysis
    # TODO 6: Select representative examples from each cluster
    # TODO 7: Generate root cause hypotheses based on cluster characteristics
    # TODO 8: Map clusters to suggested remediation strategies

    def analyze_root_causes(self, failure_cluster: FailureCluster) -> Dict[str, float]:
        """
        Analyze potential root causes for a failure cluster.

        Returns:
            Dictionary mapping root cause categories to confidence scores
        """
        # TODO 1: Analyze input characteristics of failed cases
        # TODO 2: Compare failure patterns to known failure taxonomy
        # TODO 3: Look for systematic patterns in input/output relationships
        # TODO 4: Generate confidence scores for different root cause hypotheses
        # TODO 5: Return ranked root cause possibilities

    def generate_improvement_recommendations(self, clusters: List[FailureCluster]) -> List[str]:
        """
        Generate prioritized recommendations for addressing failure patterns.

```

```
Returns:
```

```
    List of improvement recommendations prioritized by impact
```

```
"""
```

```
# TODO 1: Analyze cluster sizes and severity to prioritize improvements

# TODO 2: Map failure patterns to known remediation strategies

# TODO 3: Consider implementation difficulty vs expected impact

# TODO 4: Generate specific, actionable improvement recommendations

# TODO 5: Rank recommendations by expected impact and feasibility
```

Report Generation Implementation

```
from pathlib import Path  
  
from typing import Dict, List, Optional  
  
from jinja2 import Environment, FileSystemLoader  
  
import plotly.graph_objects as go  
  
import plotly.express as px  
  
from weasyprint import HTML  
  
import pandas as pd  
  
  
class ReportGenerator:  
  
    """Comprehensive evaluation report generation system."""  
  
  
    def __init__(self, template_dir: Path):  
        self.template_env = Environment(loader=FileSystemLoader(template_dir))  
        self.chart_config = {"displayModeBar": False, "responsive": True}  
  
  
    def generate_executive_report(self,  
        evaluation_run: EvaluationRun,  
        regression_alerts: List[RegressionAlert],  
        failure_analysis: List[FailureCluster]) -> str:  
        """  
        Generate executive summary report focused on business impact.  
  
        Returns:  
        HTML string for executive report  
        """  
  
        # TODO 1: Compute high-level performance metrics and trends  
        # TODO 2: Identify top business risks from regression alerts  
        # TODO 3: Summarize most critical failure patterns  
        # TODO 4: Generate executive-appropriate visualizations (trends, scorecards)  
  
    
```

```
# TODO 5: Create actionable recommendations with resource requirements

# TODO 6: Render executive report template with computed data


def generate_technical_report(self,
                               evaluation_run: EvaluationRun,
                               detailed_analysis: Dict) -> str:
    """
    Generate detailed technical report for engineering teams.

    Returns:
        HTML string for technical report
    """
    """
    # TODO 1: Include detailed statistical analysis with confidence intervals

    # TODO 2: Provide comprehensive failure analysis with specific examples

    # TODO 3: Generate technical visualizations (distributions, scatter plots)

    # TODO 4: Include implementation-specific recommendations with code examples

    # TODO 5: Add debugging information and investigation starting points

    # TODO 6: Render technical report template with detailed data


def create_performance_visualizations(self,
                                       evaluation_run: EvaluationRun) -> Dict[str, str]:
    """
    Create interactive visualizations for performance analysis.

    Returns:
        Dictionary mapping chart names to HTML div strings
    """
    """
    # TODO 1: Create score distribution histogram with outlier highlighting

    # TODO 2: Generate category performance heatmap with drill-down capability
```

```

# TODO 3: Build trend analysis chart if historical data available

# TODO 4: Create failure pattern sunburst chart for hierarchical analysis

# TODO 5: Generate comparison matrix for multiple evaluation runs

# TODO 6: Return dictionary of chart HTML for template embedding


def export_to_pdf(self, html_content: str, output_path: Path) -> bool:
    """
    Export HTML report to PDF format.

    Returns:
        True if export successful, False otherwise
    """
    # TODO 1: Configure PDF generation settings (page size, margins, headers)

    # TODO 2: Process HTML content to ensure PDF compatibility

    # TODO 3: Generate PDF using weasyprint with error handling

    # TODO 4: Validate generated PDF file integrity

    # TODO 5: Return success/failure status

```

Milestone Checkpoint

After implementing the reporting and analysis system, verify functionality with:

1. **Statistical Analysis Validation:** Run `python -m pytest reporting/tests/test_aggregation.py -v` to verify statistical computations are correct with known test data.
2. **Regression Detection Testing:** Create evaluation results with intentional performance degradation and verify the system detects it: `python scripts/test_regression_detection.py`
3. **Report Generation Test:** Generate all report types from sample evaluation data: `python scripts/generate_sample_reports.py`. Verify reports contain expected sections, visualizations render correctly, and PDF export works.
4. **End-to-End Analysis:** Run a complete evaluation followed by full analysis: `python -m evaluation_framework evaluate --dataset sample_data --generate-reports`. Check that insights are actionable and statistically sound.

Expected outcomes:

- Score aggregation produces statistician-validated results with proper confidence intervals
- Regression detection catches intentional performance degradation with appropriate p-values

- Failure analysis clusters reveal interpretable patterns in synthetic failure data
- Generated reports are professionally formatted and contain actionable recommendations
- Interactive visualizations work correctly in web browsers

Debugging Tips

Symptom	Likely Cause	Diagnosis	Fix
Statistical significance tests always fail	Insufficient sample size or identical distributions	Check sample sizes, plot distributions	Increase data collection or adjust significance thresholds
Failure clustering produces too many tiny clusters	Embedding similarity threshold too strict	Examine silhouette scores, visualize embeddings	Adjust clustering parameters or use different similarity metric
Reports render with missing visualizations	JavaScript/plotting library errors	Check browser console, validate chart data	Fix data serialization or chart configuration
PDF export fails silently	HTML/CSS incompatible with PDF renderer	Test HTML in browser, check weasyprint logs	Simplify CSS, use PDF-compatible styling
Regression detection produces false alarms	Natural variation mistaken for regression	Examine baseline stability, check effect sizes	Increase effect size threshold or stabilize baselines

Interactions and Data Flow

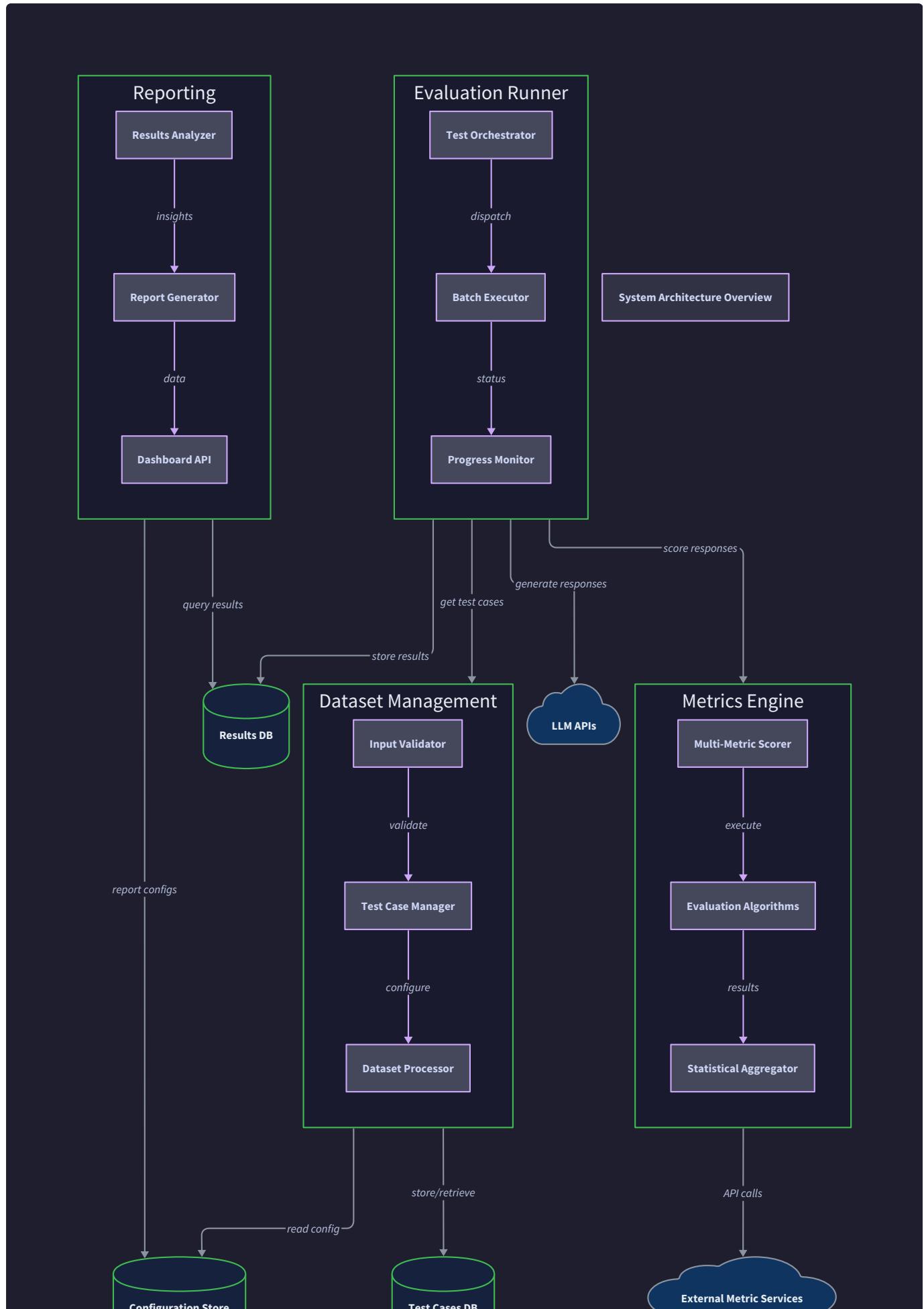
Milestone(s): All milestones (1-4) - This section describes how components from all milestones interact to complete a full evaluation, from dataset loading (Milestone 1) through metrics computation (Milestone 2), evaluation execution (Milestone 3), and final reporting (Milestone 4)

Think of the evaluation framework as a sophisticated manufacturing assembly line. Raw materials (test cases) enter at one end, pass through multiple processing stations (validation, execution, scoring, analysis), and emerge as finished products (reports with actionable insights). Each station has specific responsibilities, clear input/output contracts, and well-defined handoff protocols. Just as a factory manager needs to understand the entire flow to optimize production, developers need to understand how evaluation data flows through the system to debug issues and optimize performance.

The **evaluation sequence** represents the complete journey from dataset loading to report generation, while **component communication** defines the precise contracts between processing stations. Understanding these interactions is crucial for debugging evaluation issues, optimizing performance, and extending the framework with new capabilities.

Evaluation Sequence

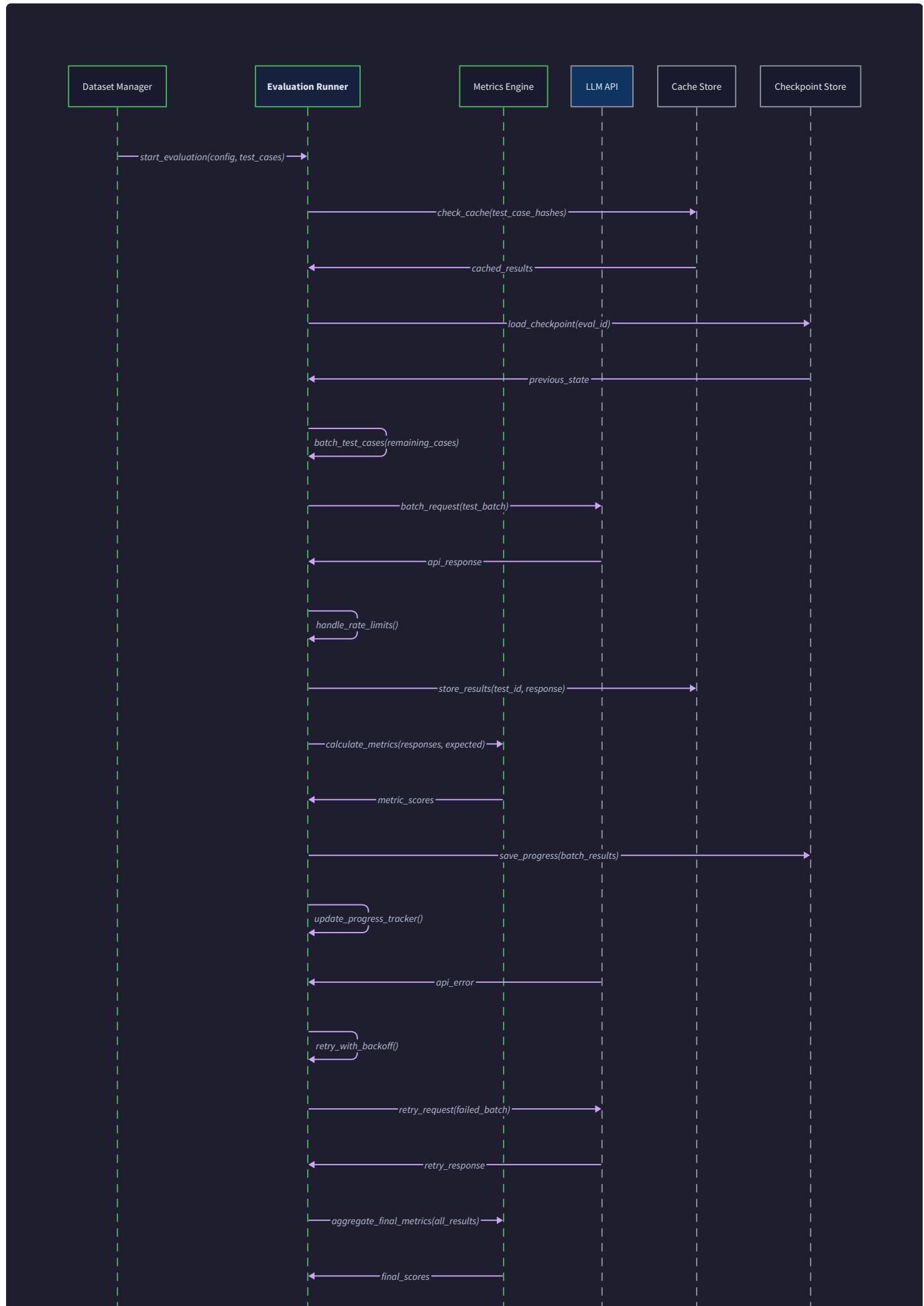
The evaluation sequence represents the complete orchestrated workflow that transforms raw test cases into actionable insights. Think of this as the "main program" that coordinates all framework components to deliver evaluation results.

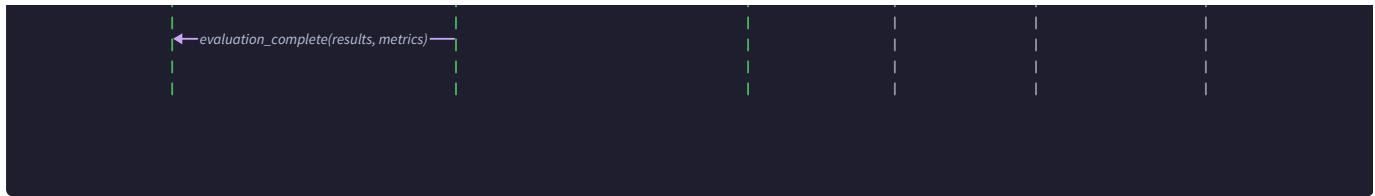


Configuration Store

Test Cases DB

Test Results DB





Phase 1: Dataset Preparation and Validation

The evaluation begins with dataset preparation, where the system ensures data quality and consistency before proceeding with expensive LLM API calls. This phase acts as a quality gate to catch issues early.

Step	Component	Action	Validation	Error Handling
1	DatasetLoader	Parse input file format (CSV/JSON/JSONL)	Schema validation against <code>TestCase</code> structure	Reject malformed files with detailed error messages
2	DatasetVersionControl	Create new version or load existing version	Content hash verification for integrity	Recovery from corrupted versions using parent fallback
3	DatasetSplitter	Apply train/test/validation splits if requested	Stratified sampling validation for balanced splits	Regenerate splits if stratification fails due to insufficient samples
4	Dataset Validator	Validate test case completeness and consistency	Check required fields (<code>prompt</code> , <code>expected_output</code> , <code>tags</code>)	Flag incomplete cases for manual review or automatic exclusion

The dataset preparation phase establishes the evaluation foundation by ensuring data quality and creating a versioned snapshot. The system performs deep validation at this stage because fixing data issues early prevents cascading failures during expensive evaluation execution.

Design Insight: Dataset validation happens before metric loading to fail fast on data issues. It's much cheaper to reject malformed test cases than to discover formatting problems after running hundreds of LLM API calls.

Phase 2: Metric Discovery and Configuration

The metrics engine discovers applicable evaluation methods and prepares the scoring pipeline. This phase determines what measurements will be taken during evaluation execution.

Step	Component	Action	Configuration	Dependencies
1	<code>MetricRegistry</code>	Discover registered metrics via plugin scanning	Load metric configurations from evaluation config	Built-in metrics (exact match, BLEU, semantic similarity)
2	Metric Validators	Check metric applicability against test cases	Validate that each metric can process the dataset schema	LLM provider APIs for LLM-as-judge metrics
3	<code>TextPreprocessor</code>	Initialize preprocessing pipelines per metric	Configure normalization rules (whitespace, case, punctuation)	No external dependencies
4	<code>EmbeddingProvider</code>	Initialize embedding models for semantic metrics	Load embedding models and warm up the cache	Embedding model files (sentence-transformers)

The metric configuration phase creates the evaluation measurement apparatus. The system validates that all requested metrics can actually process the dataset schema and that required dependencies (LLM APIs, embedding models) are available before beginning expensive evaluation execution.

Architecture Decision: Metric Applicability Checking

Decision: Pre-flight Metric Validation

- **Context:** Some metrics only apply to specific test case types (e.g., code generation metrics need code outputs)
- **Options Considered:**
 - Skip incompatible metrics silently during execution
 - Validate applicability during metric registration
 - Check applicability before evaluation execution starts
- **Decision:** Check applicability before evaluation execution starts
- **Rationale:** Failing fast prevents wasted computation and gives users clear feedback about misconfigured evaluations
- **Consequences:** Enables early error reporting but requires metrics to declare their applicability requirements

Phase 3: Evaluation Execution and Result Collection

The core evaluation phase processes test cases through parallel execution pipelines, collecting model responses and computing scores. This is the most resource-intensive phase involving LLM API calls and metric computations.

The `BatchProcessor` orchestrates parallel execution while respecting rate limits and resource constraints:

1. The batch processor reads test cases from the prepared dataset and organizes them into execution batches based on the configured `max_concurrent_evaluations` limit
2. For each batch, the system generates cache keys based on the prompt content and LLM provider configuration to check the `PromptCache` for existing responses
3. Cache misses trigger LLM API calls through the configured provider (OpenAI, Anthropic, or local inference), with the `RateLimiter` enforcing request frequency limits

4. The system stores successful LLM responses in the prompt cache with configurable TTL values to avoid redundant API calls in future evaluations
5. Each model response flows through the metrics pipeline, where applicable metrics compute scores and generate `MetricResult` objects with detailed metadata
6. The `CheckpointManager` periodically writes evaluation progress to the write-ahead log, enabling recovery from crashes or API failures
7. The `ProgressTracker` updates completion statistics and provides real-time ETA estimates based on historical processing rates

Execution Stage	Input	Processing	Output	Checkpointing
Batch Formation	<code>List[TestCase]</code>	Group by batch size and provider	<code>List[Batch]</code>	Initial batch plan
LLM Invocation	<code>TestCase</code> + provider config	API call with caching and rate limiting	Model response string	After each batch
Metric Computation	Model response + expected output	Parallel metric evaluation	<code>Dict[str, MetricResult]</code>	After metric completion
Result Aggregation	Individual metric results	Compute overall scores and metadata	<code>EvaluationResult</code>	After each test case

The evaluation execution phase implements sophisticated error handling and recovery mechanisms. The `CheckpointManager` maintains a write-ahead log that records evaluation intentions before executing them, enabling precise recovery after crashes.

Design Insight: Checkpointing happens after metric computation rather than after LLM API calls because metric computation is deterministic and cheap to recompute, while LLM responses are expensive and non-deterministic.

Common Pitfalls in Execution Phase

⚠ Pitfall: Rate Limit Cascade Failures When one LLM provider hits rate limits, naive implementations often retry immediately, causing more rate limit violations and eventually exhausting all providers. This creates a cascade failure where the entire evaluation stops making progress.

How to Avoid: Implement exponential backoff with jitter and provider health monitoring. The `RateLimiter` should track per-provider error rates and automatically reduce request frequency when rate limits are detected. The system should also distribute load across multiple providers when available.

⚠ Pitfall: Memory Exhaustion with Large Datasets Loading entire evaluation results into memory causes out-of-memory errors with large datasets. This is especially problematic when evaluation results contain large text fields or extensive metadata.

How to Avoid: Stream results to disk during evaluation and maintain only working set in memory. The `CheckpointManager` should write completed results to disk immediately and provide iterator interfaces for downstream processing.

Phase 4: Statistical Analysis and Insight Generation

The analysis phase transforms raw evaluation results into statistical insights and actionable recommendations. This phase implements the sophisticated analytics that differentiate the evaluation framework from simple scoring systems.

The analysis pipeline processes evaluation results through multiple analytical stages:

1. **Score Aggregation:** The `ScoreAggregator` computes comprehensive statistical summaries including means, medians, confidence intervals, and percentile distributions across different groupings (overall, per-tag, per-difficulty)
2. **Regression Detection:** The `RegressionDetector` compares current results against stored baselines using statistical significance testing and effect size calculations to identify performance degradation
3. **Failure Analysis:** The `FailureAnalyzer` clusters similar failure patterns using semantic similarity and identifies root cause hypotheses for systematic errors
4. **Trend Analysis:** The system analyzes performance trends over time when historical evaluation data is available, detecting changepoints where model behavior fundamentally shifted

Analysis Component	Input Data	Statistical Methods	Output Insights	Confidence Measures
<code>ScoreAggregator</code>	<code>List[EvaluationResult]</code>	Mean, median, std dev, percentiles	<code>AggregationResult</code> per category	Confidence intervals, sample sizes
<code>RegressionDetector</code>	Current + baseline results	T-tests, effect size (Cohen's d)	<code>List[RegressionAlert]</code>	P-values, statistical power
<code>FailureAnalyzer</code>	Failed evaluation results	K-means clustering on embeddings	<code>List[FailureCluster]</code>	Cluster coherence scores
Trend Analyzer	Historical evaluation runs	Changepoint detection, regression	Trend summaries and forecasts	R-squared, prediction intervals

The statistical analysis phase implements rigorous statistical methods to provide reliable insights. The system addresses multiple comparison problems by applying Bonferroni correction when testing many hypotheses simultaneously, and it reports confidence intervals alongside point estimates to communicate uncertainty.

Architecture Decision: Baseline Management Strategy

Decision: Automatic Baseline Updates with Human Approval

- **Context:** Baselines need to evolve as models improve, but automatic updates could mask regressions
- **Options Considered:**
 - Manual baseline updates only
 - Automatic baseline updates after N successful runs
 - Automatic updates with human approval workflows
- **Decision:** Automatic updates with human approval workflows
- **Rationale:** Balances baseline freshness with regression detection reliability, prevents baseline drift
- **Consequences:** Requires approval infrastructure but provides best balance of automation and safety

Phase 5: Report Generation and Export

The final phase transforms analytical insights into consumable reports tailored for different audiences. The reporting system generates both executive summaries for stakeholders and detailed technical reports for engineers.

The `ReportGenerator` orchestrates multi-format report creation:

1. **Executive Report Generation:** Creates business-focused summaries emphasizing overall performance trends, regression alerts, and high-level recommendations without technical implementation details
2. **Technical Report Generation:** Produces engineering-focused analysis with detailed metric breakdowns, failure cluster analysis, statistical test results, and specific improvement recommendations
3. **Interactive Visualization:** Generates web-based dashboards with drill-down capabilities, allowing users to explore results by filtering tags, difficulty levels, and time ranges
4. **Export Processing:** Converts HTML reports to PDF format for offline sharing and archival, maintaining formatting and embedded visualizations

Report Type	Target Audience	Key Content	Format Options	Update Frequency
Executive Summary	Business stakeholders	Trends, alerts, ROI impact	PDF, HTML email	Weekly/monthly
Technical Report	Engineers, researchers	Detailed metrics, failure analysis	Interactive HTML, PDF	Per evaluation run
Comparison Report	All audiences	Before/after analysis, A/B testing	Side-by-side HTML, PDF	On-demand
Trend Dashboard	Product managers	Historical trends, forecasting	Interactive web dashboard	Real-time

The reporting phase implements sophisticated visualization generation using statistical graphics best practices. Charts include error bars for uncertainty, use colorblind-friendly palettes, and provide alternative text descriptions for accessibility.

Design Insight: Reports are generated from the same analytical data but with different narrative structures and visual emphasis. Executive reports emphasize business impact and trends, while technical reports focus on actionable engineering improvements.

Component Communication

The component communication layer defines precise contracts and message formats that enable reliable interaction between framework subsystems. Think of this as the "API specification" that each component must implement to participate in the evaluation workflow.

Inter-Component Message Formats

Components communicate through well-defined message structures that enable loose coupling and testability. Each message type includes both data payload and metadata for debugging and auditing.

Message Type	Sender	Receiver	Payload Schema	Acknowledgment Required
DatasetLoadRequest	Evaluation Coordinator	Dataset Manager	{dataset_path: str, format: str, validation_level: str}	Yes - with validation results
DatasetLoadResponse	Dataset Manager	Evaluation Coordinator	{dataset: Dataset, validation_errors: List[str], content_hash: str}	No
MetricConfigRequest	Evaluation Coordinator	Metrics Engine	{metric_names: List[str], test_case_schema: Dict}	Yes - with applicability check
MetricConfigResponse	Metrics Engine	Evaluation Coordinator	{applicable_metrics: List[str], configuration_errors: List[str]}	No
EvaluationBatch	Evaluation Coordinator	Batch Processor	{test_cases: List[TestCase], provider_config: LLMProviderConfig, metrics: List[str]}	Yes - after completion
BatchResults	Batch Processor	Evaluation Coordinator	{results: List[EvaluationResult], batch_stats: BatchStats, errors: List[str]}	No
AnalysisRequest	Evaluation Coordinator	Analysis Engine	{evaluation_run: EvaluationRun, baseline_run_id: Optional[str]}	Yes - with analysis completion
AnalysisResponse	Analysis Engine	Evaluation Coordinator	{aggregation_results: Dict, regression_alerts: List[RegressionAlert], failure_clusters: List[FailureCluster]}	No

Error Propagation and Recovery Protocols

The framework implements structured error propagation that preserves context and enables automated recovery. Each component follows consistent error handling patterns that facilitate debugging and system resilience.

Error Classification and Handling Strategy

Error Category	Examples	Recovery Action	Escalation Threshold
Transient API Errors	Rate limits, temporary network failures	Exponential backoff retry with jitter	3 consecutive failures
Data Validation Errors	Malformed test cases, schema mismatches	Skip invalid cases with detailed logging	10% of dataset affected
Resource Exhaustion	Out of memory, disk space full	Reduce batch size, trigger garbage collection	Immediate escalation
Configuration Errors	Invalid API keys, missing model files	Fail fast with clear error messages	Immediate escalation
Dependency Failures	External service outages, model loading failures	Failover to alternative providers/models	Service-specific thresholds

The error handling system maintains an **error context chain** that tracks the complete failure path from root cause to user-visible symptom. This enables precise debugging and automated error resolution.

Error Context Chain Example:

```

Root Cause: OpenAI API rate limit exceeded
→ LLM Provider Error: HTTP 429 response with retry-after header
→ Batch Processor Error: Failed to process 5/10 test cases in batch
→ Evaluation Runner Error: Batch processing interrupted, resuming from checkpoint
→ User-Visible: Evaluation paused for rate limit cooldown, resuming in 60 seconds

```

Component Health Monitoring and Circuit Breakers

Each major component implements health monitoring that tracks error rates, response times, and resource utilization. Circuit breakers prevent cascading failures by temporarily disabling failing components.

Component	Health Metrics	Circuit Breaker Threshold	Recovery Verification
LLM Providers	Success rate, response latency, rate limit frequency	50% error rate over 1 minute	3 consecutive successful requests
Embedding Provider	Cache hit rate, computation time, model availability	Model loading failures	Successful embedding generation
Database Storage	Query success rate, connection pool health	Connection failures > 50%	Successful read/write operations
Metrics Engine	Computation time, plugin crashes	Plugin crash rate > 10%	Successful metric computation

Recovery and Checkpoint Protocols

The framework implements comprehensive recovery protocols that ensure evaluation progress is never lost, even during catastrophic failures. The recovery system operates at multiple granularities to optimize restart performance.

The `CheckpointManager` maintains three levels of recovery state:

- Transaction Log (Write-Ahead Log):** Records every intended operation before execution, enabling precise replay after crashes
- Incremental Checkpoints:** Periodic snapshots of completed work that reduce recovery time by avoiding full replay
- Full State Snapshots:** Complete evaluation state dumps for major recovery points and debugging

Recovery Scenario	Detection Method	Recovery Action	Data Loss Risk
Process Crash	Process monitoring, health checks	Resume from latest incremental checkpoint	None - all operations logged
API Provider Outage	Circuit breaker activation	Switch to backup provider or pause evaluation	None - requests cached/queued
Storage Corruption	Content hash validation	Restore from backup, replay transaction log	Minimal - recent transactions only
Memory Exhaustion	Resource monitoring	Reduce batch size, trigger garbage collection	None - checkpoint before OOM
Network Partition	Timeout detection, connectivity checks	Local processing mode, sync after recovery	None - eventual consistency

The recovery system prioritizes **evaluation continuity** over performance. When in doubt, the system chooses the safer option that preserves work and enables recovery, even if it means slower execution.

Design Insight: The checkpoint granularity balances recovery speed against storage overhead. Too frequent checkpointing slows normal execution, while too infrequent checkpointing lengthens recovery time. The system uses adaptive checkpointing that increases frequency during unstable periods.

Asynchronous Communication Patterns

The evaluation framework employs asynchronous communication patterns to maximize throughput while maintaining system responsiveness. Components communicate through message queues and event streams rather than synchronous request-response patterns.

Event-Driven Architecture for Evaluation Progress

Event Type	Publisher	Subscribers	Event Data	Processing Guarantees
TestCaseCompleted	Batch Processor	Progress Tracker, Checkpoint Manager	{case_id: str, results: EvaluationResult, processing_time: float}	At-least-once delivery
BatchProcessed	Batch Processor	Evaluation Coordinator, Analysis Engine	{batch_id: str, success_count: int, error_count: int, batch_stats: BatchStats}	Exactly-once delivery
MetricComputationFailed	Metrics Engine	Error Handler, Evaluation Coordinator	{case_id: str, metric_name: str, error: str, retry_count: int}	At-least-once delivery
RegressionDetected	Analysis Engine	Report Generator, Alert Manager	{alert: RegressionAlert, affected_metrics: List[str], confidence: float}	Exactly-once delivery
EvaluationCompleted	Evaluation Coordinator	Report Generator, Result Store	{run_id: str, final_results: EvaluationRun, generation_timestamp: datetime}	Exactly-once delivery

The asynchronous communication system implements **backpressure handling** to prevent fast producers from overwhelming slow consumers. When downstream components cannot keep up with evaluation result generation, the system automatically reduces batch sizes and introduces delays.

Message Ordering and Consistency Guarantees

The framework provides different consistency guarantees depending on the criticality of the communication:

- **Strong Consistency:** Dataset versioning operations and baseline updates require strong consistency to prevent conflicting modifications
- **Eventual Consistency:** Progress tracking and monitoring can tolerate temporary inconsistencies for better performance
- **Causal Consistency:** Evaluation results must be processed in causal order (test case completion before batch completion before evaluation completion)

Communication Pattern	Consistency Level	Ordering Requirement	Failure Handling
Dataset Modification	Strong	Global ordering	Two-phase commit
Evaluation Execution	Causal	Per-test-case ordering	Checkpoint and retry
Progress Reporting	Eventual	No ordering required	Best-effort delivery
Error Notifications	Strong	Timestamp ordering	Guaranteed delivery
Report Generation	Causal	Results before analysis	Transaction boundaries

API Contracts and Interface Specifications

Each component exposes well-defined interfaces that specify preconditions, postconditions, and error conditions. These contracts enable independent component development and comprehensive testing.

Dataset Manager Interface Contract

Method Signature	Preconditions	Postconditions	Error Conditions
<code>load_dataset(path, format, validation_level)</code>	File exists and is readable	Dataset object with validated test cases	<code>FileNotFoundException</code> , <code>ValidationException</code> , <code>FormatException</code>
<code>save_dataset_version(dataset, commit_message)</code>	Dataset contains valid test cases	Version created with unique ID	<code>DuplicateVersionException</code> , <code>StorageException</code>
<code>compute_diff(version_a, version_b)</code>	Both versions exist in storage	VersionDiff object with change details	<code>VersionNotFoundException</code> , <code>ComputationException</code>
<code>create_train_test_split(dataset, test_size, stratify_by)</code>	Dataset has required stratification tags	Tuple of train/test datasets	<code>InsufficientSamplesException</code> , <code>StratificationException</code>

Metrics Engine Interface Contract

Method Signature	Preconditions	Postconditions	Error Conditions
<code>compute(metric_name, model_output, expected_output, test_case)</code>	Metric registered and applicable	MetricResult with score 0.0-1.0	<code>MetricNotFoundError</code> , <code>ApplicabilityError</code> , <code>ComputationError</code>
<code>register_metric(metric_instance)</code>	Metric implements BaseMetric interface	Metric available in registry	<code>DuplicateMetricError</code> , <code>InterfaceError</code>
<code>get_applicable_metrics(test_case)</code>	Test case has required schema fields	List of applicable metric names	No errors (returns empty list if none applicable)
<code>preprocess_text(text, preprocessing_config)</code>	Text is valid string, config is valid	Tuple of processed text and applied steps	<code>PreprocessingError</code> , <code>ConfigurationError</code>

Evaluation Runner Interface Contract

Method Signature	Preconditions	Postconditions	Error Conditions
<code>run_evaluation(dataset, metrics, llm_config, eval_config)</code>	All components initialized and healthy	EvaluationRun with complete results	<code>ConfigurationError</code> , <code>ResourceExhaustionError</code> , <code>ProviderError</code>
<code>resume_evaluation(run_id)</code>	Checkpoint exists for run_id	EvaluationRun resumed from checkpoint	<code>CheckpointNotFoundError</code> , <code>CorruptionError</code>
<code>get_evaluation_status(run_id)</code>	Run exists in system	Status object with progress details	<code>RunNotFoundError</code>
<code>cancel_evaluation(run_id)</code>	Run is in progress	Evaluation stopped gracefully	<code>CancellationError</code> , <code>StateError</code>

Performance Monitoring and Observability

The framework implements comprehensive observability that enables performance optimization and issue diagnosis. Each component emits structured metrics and logs that provide visibility into system behavior.

Performance Metrics Collection

Metric Category	Specific Metrics	Collection Method	Alerting Thresholds
Throughput	Test cases per minute, API calls per minute, metrics computed per second	Counter increments	< 50% of baseline
Latency	API response time, metric computation time, end-to-end evaluation time	Histogram tracking	P95 > 2x baseline
Error Rates	API failures, metric computation failures, validation errors	Error counters	> 5% for any component
Resource Usage	Memory consumption, CPU utilization, disk I/O	System monitoring	> 80% sustained
Cache Performance	Hit rate, eviction rate, storage utilization	Cache statistics	Hit rate < 70%

The observability system provides **distributed tracing** that tracks evaluation requests across component boundaries. Each trace includes timing information, error details, and context that enables performance bottleneck identification.

Debugging and Diagnostic Information

The framework generates rich diagnostic information that facilitates rapid issue resolution:

- **Request Tracing:** Complete trace of each test case evaluation from input through final scoring
- **Component State Dumps:** Detailed snapshots of component internal state for debugging complex issues
- **Performance Profiles:** CPU and memory profiles during evaluation execution to identify optimization opportunities
- **Error Context:** Rich error messages that include the complete failure context and suggested remediation steps

Diagnostic Tool	Information Provided	Use Cases	Access Method
Trace Viewer	Request flow across components	Performance debugging, error diagnosis	Web UI dashboard
State Inspector	Component internal state	Configuration debugging, state corruption	Command-line tool
Performance Profiler	CPU/memory hotspots	Optimization planning	Profiling API
Error Analyzer	Error patterns and trends	System health monitoring	Automated reports

Implementation Guidance

The evaluation framework's interaction patterns require careful coordination between asynchronous components and robust error handling. This implementation guidance provides the essential infrastructure and coordination logic needed to build a reliable evaluation system.

Technology Recommendations

Component Communication	Simple Option	Advanced Option
Message Passing	Direct method calls with async/await	Redis pub/sub with message serialization
Error Propagation	Exception chaining with context	Structured error objects with recovery hints
Progress Tracking	Simple counters with periodic logging	Event streaming with real-time dashboards
Checkpointing	JSON file snapshots	SQLite with write-ahead logging
Health Monitoring	Basic logging with error counters	Prometheus metrics with Grafana dashboards

Recommended File Structure

```
evaluation_framework/
├── coordination/
│   ├── __init__.py
│   ├── evaluation_coordinator.py      ← Main orchestration logic
│   ├── message_types.py              ← Message format definitions
│   ├── error_handling.py            ← Error propagation and recovery
│   └── health_monitor.py           ← Component health tracking
├── communication/
│   ├── __init__.py
│   ├── message_queue.py            ← Async message passing
│   ├── event_bus.py                ← Event-driven notifications
│   └── circuit_breaker.py          ← Failure isolation
├── checkpointing/
│   ├── __init__.py
│   ├── checkpoint_manager.py       ← WAL and recovery
│   ├── progress_tracker.py        ← Evaluation progress
│   └── state_serialization.py     ← State persistence
└── diagnostics/
    ├── __init__.py
    ├── trace_collector.py          ← Distributed tracing
    ├── performance_monitor.py      ← Metrics collection
    └── error_analyzer.py           ← Error pattern analysis
```

Infrastructure Starter Code

Message Queue Implementation

```
import asyncio
import json
import logging
from collections import defaultdict
from dataclasses import dataclass, asdict
from typing import Any, Callable, Dict, List, Optional
from datetime import datetime

@dataclass
class Message:
    """Base message structure for component communication."""
    message_type: str
    payload: Dict[str, Any]
    sender: str
    timestamp: datetime
    correlation_id: Optional[str] = None
    retry_count: int = 0

    class AsyncMessageQueue:
        """Simple async message queue for component communication."""

        def __init__(self, max_queue_size: int = 1000):
            self._queues: Dict[str, asyncio.Queue] = defaultdict(lambda: asyncio.Queue(max_queue_size))
            self._subscribers: Dict[str, List[Callable]] = defaultdict(list)
            self._logger = logging.getLogger(__name__)
            self._running = False

        async def publish(self, topic: str, message: Message) -> None:
            """Publish message to topic."""
            try:
                await self._queues[topic].put(message)
            except满载:
                self._logger.error(f"Queue {topic} is full. Message discarded.")


    _logger = logging.getLogger(__name__)
    _running = False

    def __init__(self, max_queue_size: int = 1000):
        self._queues: Dict[str, asyncio.Queue] = defaultdict(lambda: asyncio.Queue(max_queue_size))
        self._subscribers: Dict[str, List[Callable]] = defaultdict(list)
        self._logger = logging.getLogger(__name__)
        self._running = False

    def publish(self, topic: str, message: Message) -> None:
        """Publish message to topic."""
        try:
            await self._queues[topic].put(message)
        except满载:
            self._logger.error(f"Queue {topic} is full. Message discarded.")


    _logger = logging.getLogger(__name__)
    _running = False
```

```
        self._logger.debug(f"Published {message.message_type} to {topic}")

    except asyncio.QueueFull:

        self._logger.error(f"Queue full for topic {topic}, dropping message")

        raise


def subscribe(self, topic: str, handler: Callable[[Message], None]) -> None:

    """Subscribe to messages on topic."""

    self._subscribers[topic].append(handler)

    self._logger.info(f"Subscribed to topic {topic}")


async def start(self) -> None:

    """Start message processing."""

    self._running = True

    tasks = [
        asyncio.create_task(self._process_topic(topic))
        for topic in self._subscribers.keys()
    ]

    await asyncio.gather(*tasks)


async def stop(self) -> None:

    """Stop message processing gracefully."""

    self._running = False


async def _process_topic(self, topic: str) -> None:

    """Process messages for a specific topic."""

    queue = self._queues[topic]

    handlers = self._subscribers[topic]

    while self._running:
```

```
try:

    message = await asyncio.wait_for(queue.get(), timeout=1.0)

    for handler in handlers:

        try:

            await handler(message)

        except Exception as e:

            self._logger.error(f"Handler error for {topic}: {e}")

        queue.task_done()

    except asyncio.TimeoutError:

        continue

    except Exception as e:

        self._logger.error(f"Topic processing error for {topic}: {e}")
```

Circuit Breaker Implementation

```
import asyncio

import time

from enum import Enum

from typing import Any, Callable, Optional

from dataclasses import dataclass


class CircuitState(Enum):

    CLOSED = "closed"      # Normal operation

    OPEN = "open"          # Failing, rejecting requests

    HALF_OPEN = "half_open" # Testing if service recovered


@dataclass

class CircuitBreakerConfig:

    failure_threshold: int = 5

    success_threshold: int = 3

    timeout: float = 60.0 # seconds


class CircuitBreakerError(Exception):

    pass


class CircuitBreaker:

    """Circuit breaker pattern for component failure isolation"""


    def __init__(self, name: str, config: CircuitBreakerConfig):

        self.name = name

        self.config = config

        self.state = CircuitState.CLOSED

        self.failure_count = 0

        self.success_count = 0

        self.last_failure_time: Optional[float] = None

        self._lock = asyncio.Lock()
```

```
async def call(self, func: Callable, *args, **kwargs) -> Any:
    """Execute function with circuit breaker protection."""
    async with self._lock:
        if self.state == CircuitState.OPEN:
            if self._should_attempt_reset():
                self.state = CircuitState.HALF_OPEN
                self.success_count = 0
        else:
            raise CircuitBreakerError(f"Circuit breaker {self.name} is OPEN")

    try:
        result = await func(*args, **kwargs)
        await self._on_success()
        return result
    except Exception as e:
        await self._on_failure()
        raise

def _should_attempt_reset(self) -> bool:
    """Check if enough time has passed to attempt reset."""
    if self.last_failure_time is None:
        return False
    return time.time() - self.last_failure_time >= self.config.timeout

async def _on_success(self) -> None:
    """Handle successful operation."""
    async with self._lock:
        self.failure_count = 0
```

```
if self.state == CircuitState.HALF_OPEN:

    self.success_count += 1

    if self.success_count >= self.config.success_threshold:
        self.state = CircuitState.CLOSED


async def _on_failure(self) -> None:
    """Handle failed operation."""

    async 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
```

Core Orchestration Logic Skeleton

Evaluation Coordinator (Main Orchestration)

PYTHON

```
import asyncio

from typing import Dict, List, Optional

from datetime import datetime

from .message_types import *

from .error_handling import EvaluationError, ErrorContext

class EvaluationCoordinator:

    """Main orchestrator that coordinates the complete evaluation workflow."""

    def __init__(self, config: EvaluationConfig, message_queue: AsyncMessageQueue):

        self.config = config

        self.message_queue = message_queue

        self._active_evaluations: Dict[str, EvaluationRun] = {}

        self._setup_message_handlers()

    async def run_evaluation(
            self,
            dataset_path: str,
            metrics: List[str],
            llm_config: LLMPProviderConfig
        ) -> EvaluationRun:
        """
        Execute complete evaluation workflow from dataset loading to report generation.

        This is the main entry point that orchestrates all evaluation phases.

        """
        run_id = f"eval_{datetime.now().isoformat()}"
        # TODO 1: Create EvaluationRun object and add to active evaluations
        # TODO 2: Phase 1 - Load and validate dataset
```

```
#     - Send DatasetLoadRequest to dataset manager

#     - Wait for DatasetLoadResponse and handle validation errors

#     - If validation fails, terminate evaluation with clear error message

# TODO 3: Phase 2 - Configure metrics and check applicability

#     - Send MetricConfigRequest to metrics engine

#     - Wait for MetricConfigResponse and handle configuration errors

#     - Filter out inapplicable metrics and warn user

# TODO 4: Phase 3 - Execute evaluation with batch processing

#     - Split dataset into batches based on config.max_concurrent_evaluations

#     - For each batch, send EvaluationBatch message to batch processor

#     - Collect BatchResults and handle any processing errors

#     - Update progress tracking after each batch completion

# TODO 5: Phase 4 - Perform statistical analysis

#     - Send AnalysisRequest to analysis engine with all evaluation results

#     - Wait for AnalysisResponse with aggregated insights

#     - Handle regression alerts and failure cluster analysis

# TODO 6: Phase 5 - Generate reports

#     - Send ReportGenerationRequest to report generator

#     - Wait for ReportGenerationResponse with final reports

#     - Save reports to configured output location

# TODO 7: Clean up and finalize

#     - Remove evaluation from active evaluations

#     - Send EvaluationCompleted event to notify subscribers

#     - Return completed EvaluationRun object
```

```
pass

async def resume_evaluation(self, run_id: str) -> EvaluationRun:
    """
    Resume evaluation from checkpoint after failure.

    This method handles crash recovery by loading checkpoint state.
    """

    # TODO 1: Load checkpoint state using CheckpointManager
    # TODO 2: Validate that checkpoint is not corrupted
    # TODO 3: Determine which phase was executing when failure occurred
    # TODO 4: Resume execution from the appropriate phase
    # TODO 5: Update progress tracking to reflect recovered state

    pass

def _setup_message_handlers(self) -> None:
    """
    Set up handlers for asynchronous messages from components.
    """

    # TODO: Register message handlers for each message type
    # - DatasetLoadResponse handler
    # - MetricConfigResponse handler
    # - BatchResults handler
    # - AnalysisResponse handler
    # - Error notification handlers

    pass

async def _handle_component_error(self, error_message: Message) -> None:
```

....

Handle errors reported by components during evaluation.

Implements error recovery strategies based on error type and severity.

....

```
# TODO 1: Parse error message and extract error context
# TODO 2: Determine if error is recoverable (transient vs permanent)
# TODO 3: For transient errors, implement retry logic with exponential backoff
# TODO 4: For permanent errors, gracefully terminate evaluation
# TODO 5: Update error statistics and trigger circuit breakers if needed
# TODO 6: Send error notifications to monitoring systems
```

pass

Progress Tracking and Status Reporting

```
import asyncio

import time

from typing import Optional, Dict, Any

from dataclasses import dataclass

from datetime import datetime, timedelta


@dataclass

class ProgressStats:

    """Statistics for evaluation progress tracking."""

    completed_cases: int

    total_cases: int

    failed_cases: int

    start_time: datetime

    estimated_completion: Optional[datetime]

    current_batch_size: int

    average_case_time: float


class ProgressTracker:

    """Tracks evaluation progress and provides ETA estimates."""

    def __init__(self):

        self.start_time: Optional[float] = None

        self.completed_count = 0

        self.total_count = 0

        self.failed_count = 0

        self._case_times: List[float] = []

        self._batch_stats: Dict[str, Any] = {}



    def start_tracking(self, total_cases: int) -> None:

        """Initialize progress tracking for evaluation run."""

        # TODO 1: Record start time and total case count
```

```
# TODO 2: Initialize statistics collections (case times, batch stats)

# TODO 3: Set up periodic progress reporting (every 30 seconds)

pass

def update_progress(self, completed_cases: int, failed_cases: int = 0) -> ProgressStats:
    """
    Update progress counters and recalculate ETA.

    Returns current progress statistics including time estimates.
    """

    # TODO 1: Update completed and failed case counters

    # TODO 2: Calculate average processing time per case

    # TODO 3: Estimate time remaining based on historical performance

    # TODO 4: Account for batch size changes and rate limiting

    # TODO 5: Create and return ProgressStats object

    pass

def record_case_completion(self, case_id: str, processing_time: float) -> None:
    """
    Record timing for individual case completion.

    # TODO 1: Add processing time to historical data

    # TODO 2: Maintain sliding window of recent case times (last 100 cases)

    # TODO 3: Update average processing time calculation

    # TODO 4: Detect performance degradation trends

    pass

def estimate_time_remaining(self) -> tuple[float, float]:
```

```
"""
Estimate time remaining with confidence interval.

Returns (estimated_seconds, confidence_interval_seconds).

"""

# TODO 1: Calculate remaining case count

# TODO 2: Use exponential smoothing on historical case times

# TODO 3: Account for current batch processing rate

# TODO 4: Calculate confidence interval based on time variance

# TODO 5: Return tuple of (estimate, confidence_interval)

pass
```

Milestone Checkpoints

Milestone 1 Verification: Message Passing

```
# Test basic component communication

python -m evaluation_framework.coordination.test_message_queue

# Expected: Messages flow between components without loss

# Verify: All published messages are received by subscribers

# Check: Message ordering is preserved for critical communications
```

BASH

Milestone 2 Verification: Error Handling

```
# Test error propagation and recovery

python -m evaluation_framework.coordination.test_error_handling

# Expected: Errors include full context chain

# Verify: Circuit breakers activate on repeated failures

# Check: Component recovery works after transient errors
```

BASH

Milestone 3 Verification: Progress Tracking

```
# Test progress monitoring and ETA estimation
python -m evaluation_framework.coordination.test_progress_tracker

# Expected: Progress updates reflect actual completion
# Verify: ETA estimates become more accurate over time
# Check: Checkpoint recovery restores accurate progress state
```

BASH

Milestone 4 Verification: End-to-End Coordination

```
# Test complete evaluation workflow
python -m evaluation_framework.coordination.test_full_evaluation

# Expected: Dataset → Metrics → Execution → Analysis → Reports
# Verify: Each phase completes before next phase begins
# Check: Error in any phase triggers appropriate recovery
```

BASH

Debugging Tips

Symptom	Likely Cause	Diagnosis	Fix
Evaluation hangs indefinitely	Message queue deadlock or component not responding	Check message queue sizes and component health status	Implement message timeouts and health checks
Progress tracking shows negative ETA	Case completion times recorded incorrectly	Examine case timing logs for anomalous values	Add bounds checking to timing calculations
Components receive duplicate messages	Message retry logic not tracking acknowledgments	Check message correlation IDs and retry counters	Implement idempotent message handling
Memory usage grows continuously	Messages not being cleaned up after processing	Monitor message queue sizes over time	Add message TTL and periodic cleanup
Recovery fails after checkpoint	Checkpoint corruption or version mismatch	Validate checkpoint integrity and schema version	Implement checkpoint validation and backward compatibility

Error Handling and Edge Cases

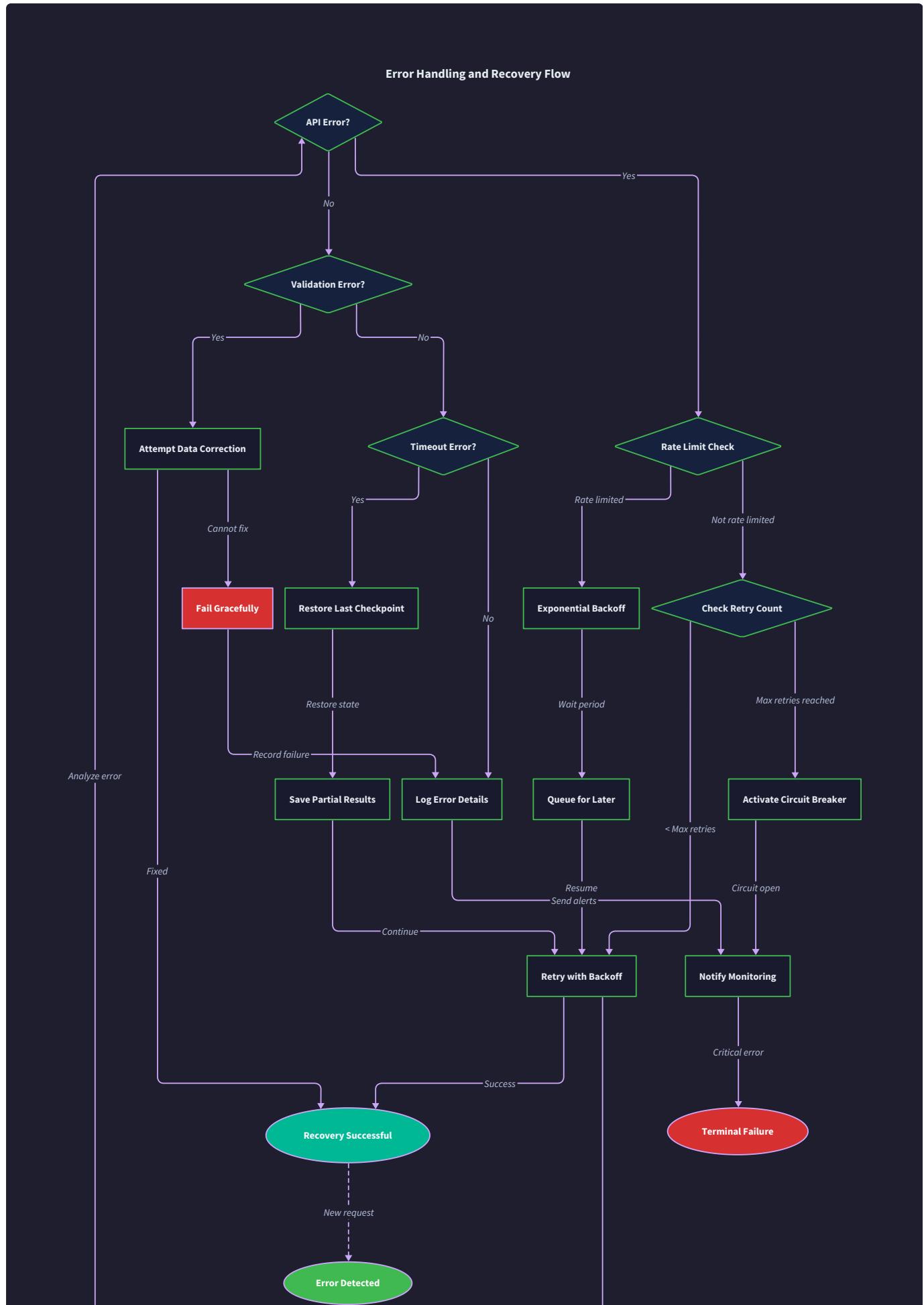
Milestone(s): All milestones (1-4) - Error handling is critical throughout dataset management (Milestone 1), metrics computation (Milestone 2), evaluation execution (Milestone 3), and reporting analysis (Milestone 4)

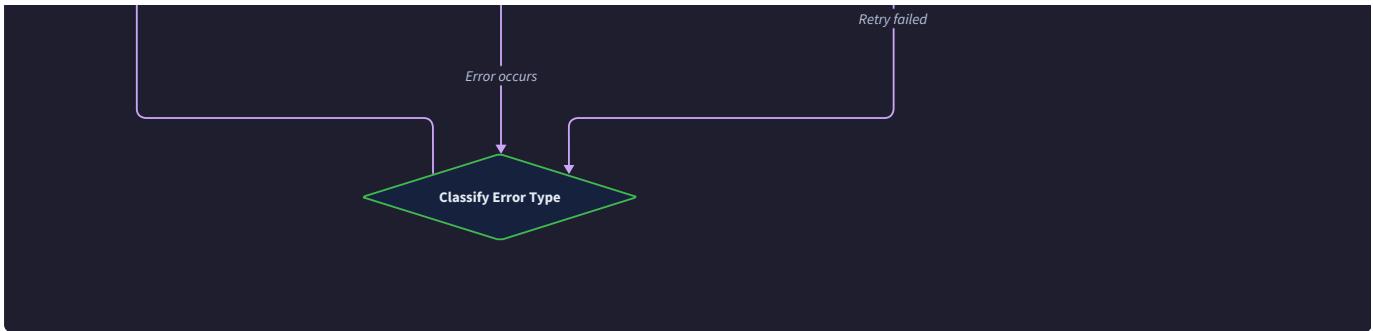
Think of error handling in an LLM evaluation framework like designing a hospital's emergency response system. Just as a hospital must handle everything from minor cuts to multi-organ failures while keeping essential services running, our

evaluation framework must gracefully handle diverse failure scenarios—from network hiccups to complete dataset corruption—while maintaining the integrity of ongoing evaluations and preserving partial progress.

The challenge in LLM evaluation systems is that errors can cascade across multiple domains simultaneously. A single API rate limit can trigger timeout exceptions, which can cause batch processing failures, which can corrupt checkpoint files, which can make evaluation recovery impossible. Unlike traditional software where errors often have local scope, evaluation framework errors tend to propagate through the entire system workflow, requiring sophisticated isolation and recovery mechanisms.

Our comprehensive error handling strategy operates on three levels: **prevention through design** (using circuit breakers, rate limiting, and validation), **graceful degradation** (failing partially while maintaining core functionality), and **recovery mechanisms** (checkpointing, retry logic, and state reconstruction). Each component maintains its own error boundaries while participating in system-wide error coordination through structured message passing and health monitoring.





Failure Modes

Understanding failure modes requires thinking like a detective—we must anticipate not just what can go wrong, but how multiple failures can interact to create cascade effects. Each failure mode has distinct characteristics in terms of **detectability** (how quickly we notice), **scope** (what gets affected), **recoverability** (whether we can fix it automatically), and **data impact** (whether we lose information).

Infrastructure Failures

Infrastructure failures represent the foundation layer where everything else depends on external services and system resources. These failures often have the highest blast radius because they affect multiple components simultaneously.

Failure Mode	Detection Method	Impact Scope	Recovery Strategy	Data Loss Risk
LLM API Rate Limiting	HTTP 429 response codes	Single provider	Exponential backoff + provider failover	None if cached
LLM API Service Outage	Connection timeouts, DNS failures	Provider-wide	Circuit breaker + alternative providers	High without caching
Network Intermittency	Socket timeouts, packet loss	Variable by connection	Retry with jitter + connection pooling	Medium
Disk Space Exhaustion	OS errno ENOSPC	System-wide writes	Cleanup + compression + external storage	Low
Memory Exhaustion	OOM killer, allocation failures	Process-level	Reduce batch sizes + garbage collection	High for in-flight work
Permission Denied	OS errno EACCES	File/directory specific	Permission repair + fallback locations	None

LLM API failures are particularly complex because they exhibit both predictable patterns (quota limits reset daily) and unpredictable behavior (model deployments causing temporary outages). Our `CircuitBreaker` component tracks failure patterns and automatically switches to alternative providers when primary services degrade.

Decision: Multi-Provider Fallback Strategy

- **Context:** LLM APIs have different availability patterns, rate limits, and performance characteristics
- **Options Considered:** Single provider with retry, round-robin across providers, intelligent failover based on health monitoring
- **Decision:** Intelligent failover with health monitoring and provider-specific circuit breakers
- **Rationale:** Different providers excel in different scenarios (OpenAI for speed, Anthropic for reasoning, local models for privacy), and their outages rarely correlate
- **Consequences:** Requires provider capability mapping and consistent response formatting, but dramatically improves evaluation reliability

Resource exhaustion failures require predictive detection rather than reactive handling. Our `ResourceMonitor` tracks disk usage, memory consumption, and file descriptor counts, triggering cleanup procedures before hard limits are reached.

Data Corruption and Validation Failures

Data corruption represents a particularly insidious class of failures because corrupted data often appears valid until it propagates through the system and causes downstream failures. Think of data corruption like a virus—by the time symptoms appear, the infection has often spread to multiple components.

Failure Mode	Detection Method	Impact Scope	Recovery Strategy	Prevention Method
Malformed JSON in Dataset	Schema validation failure	Single dataset load	Skip invalid records + error report	Pre-validation + type checking
Encoding Issues (UTF-8)	Decode exceptions	Text processing pipeline	Fallback encoding detection + normalization	Character set validation
Missing Required Fields	Field access exceptions	Test case processing	Default value substitution + warnings	Comprehensive schema validation
Circular References in Data	Infinite recursion detection	Serialization/deserialization	Reference breaking + linearization	DAG validation during construction
Corrupted Cache Files	Checksum mismatches	Cache-dependent operations	Cache invalidation + rebuilding	Content-addressable storage + checksums
Version Conflicts	Merge conflict detection	Dataset versioning	Manual resolution + conflict markers	Optimistic locking + change detection

Our `DataValidator` implements a multi-stage validation pipeline that catches corruption early. The first stage performs structural validation (JSON syntax, required fields), the second stage validates semantic constraints (score ranges, reference consistency), and the third stage performs cross-reference validation (foreign key integrity, circular dependency detection).

Validation Pipeline Flow:

1. Raw data ingestion with format detection
2. Structural validation against JSON schema
3. Field-level validation with type coercion
4. Semantic validation with business rules
5. Cross-reference validation with dependency checking
6. Repair attempts for recoverable errors
7. Error aggregation and reporting

Schema evolution failures occur when new versions of the evaluation framework encounter datasets created with older schemas. Our `SchemaUpgrader` maintains migration paths between schema versions, automatically transforming legacy data formats while preserving semantic meaning.

Concurrency and State Management Failures

Concurrency failures in evaluation systems are particularly challenging because long-running evaluations involve complex state coordination between multiple asynchronous processes. Unlike web applications where requests are short-lived, evaluation workflows can run for hours or days, creating extended windows for race conditions and state inconsistencies.

Failure Mode	Detection Method	Impact Scope	Recovery Strategy	Prevention Method
Race Conditions in Batch Processing	Inconsistent results, data races	Batch-level	Idempotent operations + retry	Pessimistic locking
Deadlock in Resource Allocation	Timeout detection, dependency cycles	Multi-resource operations	Deadlock detection + resource ordering	Lock hierarchy enforcement
Stale Cache Invalidation	Cache hit/miss ratio anomalies	Cached computations	TTL expiration + version tagging	Cache coherence protocols
Checkpoint File Corruption	Checksum validation failures	Evaluation recovery	Checkpoint rollback + partial replay	Write-ahead logging
Transaction Log Corruption	Sequential read failures	Persistent state	Log repair + state reconstruction	Redundant logging
Lost Updates in Concurrent Writes	Version mismatch detection	Shared state modifications	Optimistic concurrency control	Compare-and-swap operations

Our `CheckpointManager` implements a sophisticated write-ahead logging system that ensures evaluation state remains consistent even during system failures. Each state modification is logged before being applied, creating a recoverable transaction history.

Decision: Write-Ahead Logging for Checkpoints

- **Context:** Long-running evaluations need crash recovery without losing hours of progress
- **Options Considered:** Periodic full snapshots, incremental deltas, write-ahead logging with snapshots
- **Decision:** Hybrid approach with write-ahead logging and periodic full snapshots
- **Rationale:** WAL provides fine-grained recovery capability while snapshots bound recovery time by limiting replay length
- **Consequences:** Slightly higher I/O overhead during normal operation, but dramatically faster recovery times and stronger consistency guarantees

Evaluation Logic Failures

Evaluation logic failures represent semantic errors in the evaluation process itself—cases where the system operates correctly from a technical perspective but produces meaningless or misleading results. These failures are often the most dangerous because they can go undetected while corrupting the entire evaluation outcome.

Failure Mode	Detection Method	Impact Scope	Recovery Strategy	Prevention Method
Metric Computation Overflow	NaN/infinity detection	Individual metric results	Fallback to alternative computation	Input range validation
Division by Zero in Statistics	Exception catching	Aggregation computations	Zero-sample handling + warnings	Sample size validation
Embedding Dimensionality Mismatch	Shape validation errors	Semantic similarity metrics	Model compatibility checking	Provider validation
LLM Judge Inconsistency	Score variance analysis	LLM-as-judge evaluations	Multiple judge consensus + calibration	Judge prompt validation
Preprocessing Pipeline Errors	Text corruption detection	Text normalization	Pipeline stage isolation + rollback	Stage-by-stage validation
Custom Metric Plugin Crashes	Exception isolation		Sandboxed execution + fallback scoring	Resource limit enforcement

LLM-as-judge inconsistency is particularly challenging because it represents a fundamental reliability issue with using language models to evaluate language models. Our `JudgeCalibrator` maintains consistency by using golden examples to detect judge drift and automatically recalibrating scoring rubrics.

The **metric computation pipeline** implements defensive programming techniques where each metric computation is wrapped in error handling that provides sensible fallbacks. For example, if semantic similarity computation fails due to embedding API issues, the system falls back to fuzzy string matching with appropriate confidence scoring.

Recovery Strategies

Recovery strategies operate like a medical triage system—we must quickly assess the severity of each failure, prioritize response efforts, and apply the most effective treatment while minimizing collateral damage. The key insight is that

different failure types require fundamentally different recovery approaches, from immediate retry (for transient network issues) to complete evaluation restart (for corrupted state).

Immediate Recovery Patterns

Immediate recovery patterns handle transient failures that are likely to resolve quickly without human intervention. These patterns prioritize speed of recovery over comprehensive diagnosis, making them suitable for infrastructure hiccups and temporary resource constraints.

Exponential Backoff with Jitter forms the foundation of our retry logic. When an operation fails, we don't immediately retry—instead, we wait an exponentially increasing duration with random jitter to prevent thundering herd effects. The algorithm adapts the backoff parameters based on failure patterns, learning the optimal retry timing for different types of operations.

Retry Attempt	Base Wait Time	Jitter Range	Max Wait	Circuit Breaker Check
1	1 second	±200ms	1.2s	Closed → Continue
2	2 seconds	±400ms	2.4s	Closed → Continue
3	4 seconds	±800ms	4.8s	Half-Open → Test
4	8 seconds	±1.6s	9.6s	Open → Fail Fast
5+	16 seconds	±3.2s	19.2s	Open → Alternative Provider

Our `RetryManager` maintains separate retry policies for different operation types. API calls use aggressive retry with circuit breaker integration, while file I/O operations use conservative retry with filesystem health checks. Database operations use intermediate retry with transaction isolation validation.

Circuit Breaker Integration prevents retry storms when downstream services are genuinely unavailable. The circuit breaker tracks failure rates over sliding time windows and automatically transitions between closed (normal operation), open (fast-fail mode), and half-open (testing recovery) states.

Circuit Breaker State Transitions:

1. CLOSED: Normal operation, requests pass through
2. Failure rate exceeds threshold → transition to OPEN
3. OPEN: All requests fail immediately, preventing load on struggling service
4. After timeout period → transition to HALF_OPEN
5. HALF_OPEN: Single test request allowed through
6. Test request succeeds → CLOSED, test request fails → OPEN

Provider Failover implements intelligent routing across multiple LLM providers based on real-time health monitoring.

When the primary provider fails, the system automatically routes requests to healthy alternatives while maintaining response format consistency.

Provider Status	Routing Decision	Fallback Chain	Quality Impact
Primary Healthy	Route to primary	N/A	Baseline quality
Primary Degraded	Load balance across healthy	Secondary → Tertiary	Minimal impact
Primary Failed	Route to best alternative	Health-ranked providers	Possible quality variation
All Providers Degraded	Queue requests + batch retry	Local model fallback	Significant impact

Graceful Degradation Strategies

Graceful degradation ensures that partial failures don't bring down the entire evaluation system. Instead of treating any failure as a complete system failure, we identify which functionality can continue operating and which needs to be temporarily disabled.

Partial Evaluation Continuation allows evaluations to proceed even when some metrics fail or some test cases encounter errors. The system maintains a detailed error log while computing results for all successful operations, providing maximum value from partially successful runs.

Our `EvaluationCoordinator` implements a sophisticated dependency analysis system that determines which operations can proceed independently. If semantic similarity metrics fail due to embedding API issues, exact match and LLM-as-judge metrics can continue operating normally. The final report clearly indicates which results are complete and which are partial.

Component Failure	Continuing Operations	Degraded Operations	User Notification
Embedding API Down	Exact match, LLM-as-judge	Semantic similarity disabled	Warning in report header
Primary LLM Provider Failed	Exact match, semantic similarity	LLM-as-judge using secondary provider	Provider switch notification
Disk Nearly Full	All metrics computation	Caching disabled, reporting limited	Resource constraint warning
Network Intermittent	Cached operations	Real-time API calls queued	Network status indicator

Metric Fallback Hierarchies provide alternative scoring methods when primary metrics fail. For instance, if semantic similarity computation fails, the system can fall back to fuzzy string matching with appropriate confidence adjustments. Each fallback is clearly marked in the results to maintain evaluation transparency.

Checkpointed Progress Preservation ensures that even catastrophic failures don't result in complete work loss. The system continuously saves progress checkpoints that allow evaluation resumption from the last successful batch, minimizing re-computation requirements.

Decision: Granular Checkpoint Strategy

- **Context:** Long-running evaluations need fine-grained recovery without excessive I/O overhead
- **Options Considered:** Checkpoint after every test case, checkpoint after every batch, checkpoint only at major milestones
- **Decision:** Adaptive checkpointing based on evaluation progress and system health
- **Rationale:** Frequent checkpoints during unstable periods, less frequent during stable operation, balancing recovery granularity with performance overhead
- **Consequences:** More complex checkpoint management logic, but optimal balance between recovery capability and system performance

State Reconstruction and Repair

State reconstruction handles scenarios where system state has become corrupted or inconsistent, requiring careful analysis and repair procedures. Unlike immediate recovery patterns that assume the system is fundamentally healthy, state reconstruction acknowledges that something has gone fundamentally wrong and needs to be fixed.

Transaction Log Replay reconstructs consistent system state from write-ahead logs when checkpoint files become corrupted. The replay process validates each logged operation before applying it, skipping corrupted or inconsistent entries while maintaining referential integrity.

The replay algorithm operates in several phases:

1. **Log Validation Phase:** Scan the entire transaction log to identify corrupted entries, missing sequences, and consistency violations
2. **Checkpoint Selection Phase:** Identify the latest valid checkpoint that can serve as a replay starting point
3. **Incremental Replay Phase:** Apply logged operations in strict chronological order, validating preconditions before each operation
4. **Consistency Verification Phase:** Verify that the reconstructed state satisfies all system invariants and cross-reference constraints
5. **Reconciliation Phase:** Handle any remaining inconsistencies through conflict resolution rules or user intervention prompts

Cache Coherence Restoration handles scenarios where cached results have become inconsistent with their underlying data sources. Our cache system maintains content-addressable storage where cache keys are derived from input hashes, but cache corruption can still occur due to file system issues or concurrent access problems.

Cache Inconsistency Type	Detection Method	Repair Strategy	Prevention Update
Stale Embeddings	Embedding model version mismatch	Batch recomputation + versioning	Model version tracking
Corrupted Response Cache	Checksum validation failure	Entry eviction + API retry	Atomic write operations
Orphaned Cache Entries	Reference counting audit	Garbage collection + compaction	Reference tracking
Cache Size Explosion	Disk usage monitoring	LRU eviction + size limits	Proactive cleanup policies

Dataset Version Reconciliation resolves conflicts when multiple users simultaneously modify evaluation datasets. Our version control system detects conflicts at the test case level and provides structured conflict resolution tools.

When merge conflicts occur, the system presents users with a three-way diff view showing the base version, both conflicting modifications, and suggested resolutions. For automated resolution, we implement conflict resolution rules:

- Content conflicts (same test case modified differently) require manual resolution
- Addition conflicts (same case_id added with different content) prioritize the modification with more comprehensive metadata
- Tag conflicts (same test case tagged differently) merge tag sets with conflict annotations
- Metadata conflicts (same test case with different difficulty/category) use most recent timestamp

Error Propagation and Isolation

Error propagation control prevents single component failures from cascading throughout the entire system. We implement bulkhead patterns that isolate different types of operations while maintaining coordinated error reporting and recovery.

Component Isolation Boundaries ensure that failures in one component don't directly affect others. Each major component (Dataset Manager, Metrics Engine, Evaluation Runner, Reporter) operates with its own error handling context and resource limits.

Our `ErrorBoundary` system intercepts exceptions at component boundaries and translates them into structured error messages that other components can understand and respond to appropriately. This prevents the error handling logic of one component from needing deep knowledge about the internal failure modes of other components.

Component	Isolation Mechanism	Error Translation	Recovery Coordination
Dataset Manager	Process isolation + file locks	Dataset operation errors → structured messages	Version rollback coordination
Metrics Engine	Thread pools + resource limits	Metric computation errors → score invalidation	Partial result aggregation
Evaluation Runner	Async task isolation + timeouts	Execution errors → batch failure notifications	Checkpoint coordination
Reporter	Memory isolation + temp files	Report generation errors → partial output	Alternative format generation

Structured Error Context Propagation ensures that when errors do cross component boundaries, they carry sufficient context for intelligent handling decisions. Each error includes not just the immediate failure reason, but also the evaluation context, attempted recovery actions, and suggested next steps.

Our `ErrorHandler` data structure captures:

- **Failure Classification:** Transient vs. permanent, local vs. systemic, recoverable vs. fatal
- **Operational Context:** Which evaluation run, which test cases, which metrics, which providers
- **Environmental Context:** System resources, network conditions, concurrent operations
- **Recovery History:** What recovery attempts have been made, what worked, what failed
- **Impact Assessment:** What functionality is affected, what can continue, what must stop

This rich error context enables sophisticated recovery decision making. For example, if embedding API failures are detected during semantic similarity computation, the error context indicates whether this affects a small subset of test cases (continue with warnings) or the majority of the evaluation (pause and retry with different provider).

Implementation Guidance

The error handling implementation bridges robust theoretical design with practical engineering realities. Junior developers often struggle with error handling because it requires thinking about all the ways things can go wrong—which requires experience they don't yet have. Our implementation provides comprehensive error handling infrastructure that catches common mistakes while being extensible for domain-specific failure modes.

Technology Recommendations

Component	Simple Option	Advanced Option
Retry Logic	<code>tenacity</code> library with basic exponential backoff	Custom retry manager with adaptive backoff and circuit breakers
Circuit Breaker	<code>pybreaker</code> library	Custom implementation with health monitoring integration
Logging	Python <code>logging</code> module with structured formatters	<code>structlog</code> with JSON output and distributed tracing
Error Tracking	File-based error logs with rotation	<code>sentry-sdk</code> for error aggregation and alerting
Health Monitoring	Simple heartbeat checks	<code>prometheus-client</code> with custom metrics
State Persistence	<code>pickle</code> for checkpoint serialization	<code>sqlalchemy</code> with transaction support

File Structure Organization

```
evaluation_framework/
├── core/
│   ├── errors/
│   │   ├── __init__.py           ← Error type definitions
│   │   ├── handlers.py          ← Error handler implementations
│   │   ├── recovery.py          ← Recovery strategy implementations
│   │   ├── circuit_breaker.py   ← Circuit breaker component
│   │   └── retry_manager.py     ← Retry logic with backoff
│   ├── monitoring/
│   │   ├── health_monitor.py    ← Component health tracking
│   │   ├── error_aggregator.py  ← Error pattern analysis
│   │   └── alerts.py            ← Error notification system
│   └── persistence/
│       ├── checkpoint_manager.py ← Write-ahead logging and recovery
│       ├── transaction_log.py   ← Durable state tracking
│       └── cache_manager.py     ← Cache consistency management
├── dataset/
│   ├── validation/
│   │   ├── schema_validator.py  ← Data structure validation
│   │   ├── content_validator.py  ← Semantic validation rules
│   │   └── repair_strategies.py ← Automatic data repair
└── evaluation/
    ├── execution/
    │   ├── error_isolation.py    ← Component boundary management
    │   ├── partial_evaluation.py  ← Graceful degradation logic
    │   └── recovery_coordinator.py ← Cross-component recovery
```

Infrastructure Starter Code

Here's a complete error handling foundation that handles the complex infrastructure concerns so learners can focus on domain-specific error scenarios:

Circuit Breaker Implementation (`core/errors/circuit_breaker.py`):

```
import time

import threading

from enum import Enum

from typing import Optional, Callable, Any

from dataclasses import dataclass


class CircuitState(Enum):

    CLOSED = "closed"

    OPEN = "open"

    HALF_OPEN = "half_open"

    @dataclass

    class CircuitBreakerConfig:

        failure_threshold: int = 5

        success_threshold: int = 2

        timeout: float = 60.0

        expected_exceptions: tuple = (Exception,)

    class CircuitBreaker:

        """Circuit breaker implementation with automatic failure detection and recovery."""

        def __init__(self, name: str, config: CircuitBreakerConfig):

            self.name = name

            self.config = config

            self.state = CircuitState.CLOSED

            self.failure_count = 0

            self.success_count = 0

            self.last_failure_time: Optional[float] = None

            self._lock = threading.RLock()

        def call(self, func: Callable, *args, **kwargs) -> Any:
```

```

"""Execute function with circuit breaker protection."""

with self._lock:

    if self.state == CircuitState.OPEN:

        if self._should_attempt_reset():

            self.state = CircuitState.HALF_OPEN

            self.success_count = 0

    else:

        raise CircuitBreakerOpenException(f"Circuit breaker {self.name} is OPEN")


try:

    result = func(*args, **kwargs)

    self._on_success()

    return result

except self.config.expected_exceptions as e:

    self._on_failure()

    raise e


def _should_attempt_reset(self) -> bool:

    """Check if enough time has passed to attempt circuit breaker reset."""

    if self.last_failure_time is None:

        return True

    return (time.time() - self.last_failure_time) >= self.config.timeout


def _on_success(self) -> None:

    """Handle successful circuit breaker operation."""

    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 _on_failure(self) -> None:

        """Handle failed circuit breaker operation."""

        self.failure_count += 1

        self.last_failure_time = time.time()

        if self.failure_count >= self.config.failure_threshold:

            self.state = CircuitState.OPEN

    class CircuitBreakerOpenException(Exception):

        """Exception raised when circuit breaker is in OPEN state."""

        pass
```

Retry Manager with Adaptive Backoff (core/errors/retry_manager.py):

```
import random

import time

import logging

from typing import Optional, Callable, Any, Type

from dataclasses import dataclass


@dataclass

class RetryConfig:

    max_attempts: int = 3

    base_delay: float = 1.0

    max_delay: float = 60.0

    exponential_base: float = 2.0

    jitter: bool = True

    retriable_exceptions: tuple = (Exception,)

class RetryManager:

    """Retry manager with exponential backoff and jitter."""

    def __init__(self, config: RetryConfig):

        self.config = config

        self.logger = logging.getLogger(__name__)

    def execute_with_retry(self, func: Callable, *args, **kwargs) -> Any:

        """Execute function with retry logic and exponential backoff."""

        last_exception = None

        for attempt in range(self.config.max_attempts):

            try:

                result = func(*args, **kwargs)

                if attempt > 0:

                    self.logger.info(f"Function succeeded on attempt {attempt + 1}")

            except Exception as e:
```

```
        return result

    except self.config.retryable_exceptions as e:
        last_exception = e

        if attempt < self.config.max_attempts - 1:
            delay = self._calculate_delay(attempt)
            self.logger.warning(
                f"Attempt {attempt + 1} failed: {e}. Retrying in {delay:.2f}s"
            )
            time.sleep(delay)
        else:
            self.logger.error(f"All {self.config.max_attempts} attempts failed")

    # All attempts failed
    raise last_exception

def _calculate_delay(self, attempt: int) -> float:
    """Calculate delay for exponential backoff with jitter."""
    delay = min(
        self.config.base_delay * (self.config.exponential_base ** attempt),
        self.config.max_delay
    )

    if self.config.jitter:
        # Add random jitter (±25% of delay)
        jitter_range = delay * 0.25
        delay += random.uniform(-jitter_range, jitter_range)
        delay = max(0.1, delay) # Ensure minimum delay
```

```
return delay
```

Checkpoint Manager with Write-Ahead Logging (`core/persistence/checkpoint_manager.py`):

```
import json

import os

import hashlib

import time

from pathlib import Path

from typing import Optional, Dict, Any, List

from dataclasses import dataclass, asdict


@dataclass

class CheckpointEntry:

    timestamp: float

    operation_type: str

    data: Dict[str, Any]

    checksum: str


class CheckpointManager:

    """Checkpoint manager with write-ahead logging for crash recovery."""

    def __init__(self, checkpoint_dir: Path):

        self.checkpoint_dir = Path(checkpoint_dir)

        self.checkpoint_dir.mkdir(parents=True, exist_ok=True)

        self.wal_path = self.checkpoint_dir / "write_ahead.log"

        self.last_full_snapshot = 0

        self._ensure_wal_exists()

    def _ensure_wal_exists(self) -> None:

        """Ensure write-ahead log file exists."""

        if not self.wal_path.exists():

            self.wal_path.touch()

    def write_incremental_checkpoint(self, operation_type: str, data: Dict[str, Any]) -> None:
```

```
"""Write incremental progress checkpoint to WAL."""

entry = CheckpointEntry(
    timestamp=time.time(),
    operation_type=operation_type,
    data=data,
    checksum=self._calculate_checksum(data)
)

# Append to write-ahead log

with open(self.wal_path, 'a', encoding='utf-8') as f:
    f.write(json.dumps(asdict(entry)) + '\n')
    f.flush()
    os.fsync(f.fileno()) # Force write to disk


def write_full_snapshot(self, evaluation_run: 'EvaluationRun') -> None:
    """Write complete state snapshot."""

    snapshot_path = self.checkpoint_dir / f"snapshot_{int(time.time())}.json"
    snapshot_data = evaluation_run.to_dict()

    with open(snapshot_path, 'w', encoding='utf-8') as f:
        json.dump(snapshot_data, f, indent=2)
        f.flush()
        os.fsync(f.fileno())

    self.last_full_snapshot = time.time()

    self._cleanup_old_snapshots()


def recover_evaluation_state(self) -> Optional['EvaluationRun']:
    """Recover evaluation state from checkpoints and WAL."""
```

```
# Find latest snapshot

latest_snapshot = self._find_latest_snapshot()

if not latest_snapshot:

    return None


# Load base state from snapshot

with open(latest_snapshot, 'r', encoding='utf-8') as f:

    base_state = json.load(f)


# Apply WAL entries since snapshot

wal_entries = self._read_wal_since_snapshot(latest_snapshot)

recovered_state = self._apply_wal_entries(base_state, wal_entries)


# Reconstruct EvaluationRun object

from ..data_model import EvaluationRun

return EvaluationRun.from_dict(recovered_state)


def _calculate_checksum(self, data: Dict[str, Any]) -> str:

    """Calculate checksum for data integrity verification."""

    data_str = json.dumps(data, sort_keys=True)

    return hashlib.sha256(data_str.encode()).hexdigest()


def _find_latest_snapshot(self) -> Optional[Path]:

    """Find the most recent snapshot file."""

    snapshots = list(self.checkpoint_dir.glob("snapshot_*.json"))

    if not snapshots:

        return None

    return max(snapshots, key=lambda p: p.stat().st_mtime)
```

```
def _read_wal_since_snapshot(self, snapshot_path: Path) -> List[CheckpointEntry]:  
  
    """Read WAL entries created since the snapshot."""  
  
    snapshot_time = snapshot_path.stat().st_mtime  
  
    entries = []  
  
  
    try:  
  
        with open(self.wal_path, 'r', encoding='utf-8') as f:  
  
            for line in f:  
  
                if line.strip():  
  
                    entry_data = json.loads(line)  
  
                    entry = CheckpointEntry(**entry_data)  
  
                    if entry.timestamp > snapshot_time:  
  
                        # Verify checksum  
  
                        expected_checksum = self._calculate_checksum(entry.data)  
  
                        if entry.checksum == expected_checksum:  
  
                            entries.append(entry)  
  
                        else:  
  
                            logging.warning(f"Corrupt WAL entry detected, skipping: {entry}")  
  
    except (FileNotFoundException, json.JSONDecodeError) as e:  
  
        logging.error(f"Error reading WAL: {e}")  
  
  
    return entries  
  
  
def _apply_wal_entries(self, base_state: Dict[str, Any], entries: List[CheckpointEntry]) -> Dict[str, Any]:  
  
    """Apply WAL entries to reconstruct current state."""  
  
    current_state = base_state.copy()  
  
  
    for entry in entries:  
  
        if entry.operation_type == "test_case_completed":
```

```

        # Add completed test case result

        if "results" not in current_state:
            current_state["results"] = []
        current_state["results"].append(entry.data["result"])

    elif entry.operation_type == "batch_processed":

        # Update batch processing statistics

        if "summary_stats" not in current_state:
            current_state["summary_stats"] = {}
        current_state["summary_stats"].update(entry.data["stats"])

    return current_state

def _cleanup_old_snapshots(self, keep_count: int = 5) -> None:
    """Clean up old snapshot files, keeping only the most recent ones."""
    snapshots = sorted(
        self.checkpoint_dir.glob("snapshot_*.json"),
        key=lambda p: p.stat().st_mtime,
        reverse=True
    )

    for old_snapshot in snapshots[keep_count:]:
        old_snapshot.unlink()

```

Core Logic Skeleton Code

Error Handler Implementation (`core/errors/handlers.py`):

```
from typing import Dict, Any, Optional, Callable, List

from enum import Enum

from dataclasses import dataclass

import logging

class ErrorSeverity(Enum):

    LOW = "low"

    MEDIUM = "medium"

    HIGH = "high"

    CRITICAL = "critical"

    @dataclass

    class ErrorContext:

        """Rich error context for intelligent recovery decisions."""

        error_type: str

        severity: ErrorSeverity

        component: str

        operation: str

        evaluation_context: Dict[str, Any]

        recoverySuggestions: List[str]

        metadata: Dict[str, Any]

    class ErrorHandler:

        """Central error handling coordinator with recovery strategy selection."""

        def __init__(self):

            self.logger = logging.getLogger(__name__)

            self.recovery_strategies: Dict[str, Callable] = {}

            self.error_patterns: Dict[str, int] = {}

        def handle_error(self, exception: Exception, context: ErrorContext) -> bool:
```

```
"""
Handle error with appropriate recovery strategy.

Returns True if recovery was successful, False if error is fatal.

"""

# TODO 1: Log error with full context for debugging and monitoring

# TODO 2: Classify error type and determine if it's recoverable

# TODO 3: Check error patterns to detect systemic issues (e.g., repeated API failures)

# TODO 4: Select appropriate recovery strategy based on error type and context

# TODO 5: Attempt recovery using selected strategy

# TODO 6: Update error pattern tracking for future handling decisions

# TODO 7: Return recovery success status

# Hint: Use self._classify_error(exception) to determine error category

# Hint: Use self._select_recovery_strategy(error_type, context) for strategy selection

pass


def register_recovery_strategy(self, error_type: str, strategy: Callable) -> None:

    """Register a recovery strategy for specific error types."""

    # TODO 1: Validate that strategy is callable and has correct signature

    # TODO 2: Store strategy in registry with error type as key

    # TODO 3: Log strategy registration for debugging

    pass


def _classify_error(self, exception: Exception) -> str:

    """
    Classify exception into error categories for recovery strategy selection.

    Categories: 'network', 'api_rate_limit', 'validation', 'resource', 'logic'
    """

    # TODO 1: Check exception type and message patterns

    # TODO 2: Map common exception types to error categories
```

```
# TODO 3: Return appropriate category string

# Hint: Check for requests.exceptions for network errors

# Hint: Look for HTTP 429 status codes for rate limiting

# Hint: Check for ValidationError types for data validation issues

pass

def _select_recovery_strategy(self, error_type: str, context: ErrorContext) ->
Optional[Callable]:
    """Select the most appropriate recovery strategy for given error and context."""

    # TODO 1: Look up registered recovery strategies for error type

    # TODO 2: Consider error severity and component context

    # TODO 3: Check recent error patterns to avoid ineffective strategies

    # TODO 4: Return best strategy or None if no recovery possible

    pass
```

Recovery Strategy Implementations (`core/errors/recovery.py`):

```
from typing import Dict, Any, Optional, List

import time

import logging

from .circuit_breaker import CircuitBreaker, CircuitBreakerConfig

from .retry_manager import RetryManager, RetryConfig

class RecoveryStrategies:

    """Collection of recovery strategies for different error types."""

    def __init__(self):
        self.logger = logging.getLogger(__name__)
        self.circuit_breakers: Dict[str, CircuitBreaker] = {}
        self.retry_managers: Dict[str, RetryManager] = {}

    def recover_api_failure(self, context: ErrorContext) -> bool:
        """
        Recover from LLM API failures using provider failover and circuit breakers.

        """
        # TODO 1: Extract provider information from error context
        # TODO 2: Check circuit breaker status for failed provider
        # TODO 3: If circuit is open, attempt failover to alternative provider
        # TODO 4: Update provider health status based on recovery success
        # TODO 5: Return True if recovery successful, False otherwise
        # Hint: Use self._get_circuit_breaker(provider) to get provider-specific breaker
        # Hint: Check context.evaluation_context for available alternative providers
        pass

    def recover_resource_exhaustion(self, context: ErrorContext) -> bool:
        """
        Recover from resource exhaustion by reducing batch sizes and cleaning up.
        """
```

```
"""
# TODO 1: Identify exhausted resource type (memory, disk, file descriptors)

# TODO 2: Trigger appropriate cleanup procedures (cache cleanup, temp file removal)

# TODO 3: Reduce resource usage parameters (batch sizes, concurrent operations)

# TODO 4: Wait for resource recovery before continuing operations

# TODO 5: Return recovery success status

# Hint: Check context.metadata for current resource usage levels

# Hint: Use exponential backoff for resource availability checking

pass

def recover_data_corruption(self, context: ErrorContext) -> bool:
"""
Recover from data corruption by repairing or reloading affected data.

"""

# TODO 1: Assess scope of data corruption from error context

# TODO 2: Attempt automatic repair for recoverable corruption types

# TODO 3: For unrecoverable corruption, reload from authoritative source

# TODO 4: Validate repaired/reloaded data before continuing

# TODO 5: Update data integrity monitoring to prevent similar issues

# Hint: Use checksums and content hashes to verify data integrity

# Hint: Maintain backup copies of critical datasets for recovery

pass

def recover_state_inconsistency(self, context: ErrorContext) -> bool:
"""
Recover from state inconsistency by reconstructing consistent state.

"""

# TODO 1: Analyze inconsistency type and affected state components

# TODO 2: Identify authoritative source for consistent state reconstruction
```

```
# TODO 3: Rebuild inconsistent state from transaction logs or snapshots

# TODO 4: Verify state consistency across all affected components

# TODO 5: Resume operations from consistent state

# Hint: Use write-ahead logs to replay operations and rebuild state

# Hint: Implement state validation checks to verify consistency

pass
```

Milestone Checkpoints

Checkpoint 1: Basic Error Handling Infrastructure

- **Expected Command:** `python -m pytest tests/test_error_handling.py::TestBasicErrorHandler -v`
- **Expected Output:** All basic error handler tests pass, including exception classification, recovery strategy registration, and error context creation
- **Manual Verification:** Create a simple test script that intentionally triggers different error types (network failure, validation error, resource constraint) and verify that appropriate error classifications and recovery suggestions are generated
- **Success Indicators:** Error handler correctly classifies 5+ different exception types, recovery strategies can be registered and retrieved, error contexts contain all required fields
- **Troubleshooting:** If error classification fails, check that exception type checking logic covers both exception class and message pattern matching

Checkpoint 2: Circuit Breaker and Retry Logic

- **Expected Command:** `python -m pytest tests/test_circuit_breaker.py tests/test_retry_manager.py -v`
- **Expected Output:** Circuit breaker state transitions work correctly, retry logic implements exponential backoff with jitter
- **Manual Verification:** Create a mock API that fails intermittently and verify that circuit breaker opens after threshold failures, then closes after successful recovery period
- **Success Indicators:** Circuit breaker prevents cascading failures, retry logic adapts backoff timing based on failure patterns, integration between circuit breaker and retry manager works smoothly
- **Troubleshooting:** If circuit breaker doesn't open, verify that failure counting logic correctly tracks consecutive failures vs. total failures over time window

Checkpoint 3: Checkpoint Recovery System

- **Expected Command:** `python -m pytest tests/test_checkpoint_manager.py::TestRecoveryFlow -v`
- **Expected Output:** Write-ahead logging correctly records operations, recovery process successfully reconstructs state from checkpoints
- **Manual Verification:** Start a long-running evaluation, kill the process midway, restart and verify that evaluation resumes from last checkpoint with no lost progress
- **Success Indicators:** WAL entries have correct checksums, snapshot files are created periodically, recovery process handles corrupted log entries gracefully

- **Troubleshooting:** If recovery fails, check that WAL entries are being flushed to disk (fsync) and that JSON serialization handles all data types in evaluation state

Checkpoint 4: End-to-End Error Scenarios

- **Expected Command:** `python scripts/test_error_scenarios.py --scenario all`
- **Expected Output:** System gracefully handles API outages, data corruption, resource exhaustion, and concurrent access conflicts
- **Manual Verification:** Run evaluation with intentionally unreliable infrastructure (rate-limited APIs, low disk space, intermittent network) and verify that partial results are preserved and evaluation can complete
- **Success Indicators:** System maintains partial progress during failures, error reports provide actionable information for debugging, recovery procedures minimize data loss
- **Troubleshooting:** If cascading failures occur, verify that error boundaries are properly isolating component failures and that circuit breakers are preventing retry storms

Debugging Tips

Symptom	Likely Cause	Diagnosis Method	Fix
Evaluation hangs indefinitely	Deadlock in resource allocation or infinite retry loop	Check thread dumps, examine lock acquisition order, review retry configuration	Implement timeout limits, enforce lock ordering hierarchy, add circuit breaker to retry logic
Partial results lost after crash	Checkpoint frequency too low or WAL not syncing to disk	Check checkpoint timestamps, verify fsync calls in WAL writing	Increase checkpoint frequency during unstable periods, ensure all file writes use fsync
Circuit breaker stuck in OPEN state	Success threshold too high or test requests failing	Monitor circuit breaker state transitions, check health of alternative providers	Lower success threshold, verify alternative provider configuration, implement manual circuit reset
Error recovery attempts consume excessive resources	Recovery strategies not respecting resource limits	Monitor resource usage during recovery, check for exponential backoff implementation	Add resource usage tracking to recovery strategies, implement adaptive backoff based on system load
Inconsistent evaluation results after recovery	State reconstruction not maintaining referential integrity	Compare recovered state with expected state, validate cross-references	Add comprehensive state validation after recovery, implement referential integrity checks
Recovery strategies not being triggered	Error classification not matching registered strategy types	Log error classification results, verify strategy registration	Fix error classification logic to handle edge cases, add fallback strategies for unclassified errors

Testing Strategy

Milestone(s): All milestones (1-4) - Testing strategy spans dataset management (Milestone 1), metrics computation (Milestone 2), evaluation execution (Milestone 3), and reporting analysis (Milestone 4)

Testing an LLM evaluation framework presents unique challenges compared to traditional software testing. Think of it like quality assurance for a quality assurance system - we need to verify that our system correctly measures the quality of something inherently subjective and probabilistic. Unlike testing a calculator where $2+2$ always equals 4, we're testing a system where "good" output depends on context, domain expertise, and often human judgment. Our testing strategy must therefore balance mathematical precision with the inherent fuzziness of language evaluation.

The testing approach follows a **test pyramid architecture** with three distinct layers, each serving different validation purposes. At the foundation, unit tests verify individual components work correctly in isolation - can our exact match metric correctly identify identical strings? In the middle, integration tests verify components work together properly - does our metrics engine correctly orchestrate multiple metrics and aggregate their results? At the apex, end-to-end tests verify the complete evaluation workflow produces reliable, actionable insights for real-world scenarios.

Each milestone introduces specific testing challenges that require specialized approaches. Dataset management testing must verify data integrity across versions and formats. Metrics testing must validate both deterministic scoring (exact match) and probabilistic scoring (semantic similarity, LLM-as-judge). Evaluation runner testing must verify correct behavior under concurrency, failures, and resource constraints. Reporting testing must validate statistical computations and ensure insights are mathematically sound.

Test Pyramid

The test pyramid organizes our testing strategy into three layers with different scopes, speeds, and purposes. This structure ensures comprehensive coverage while maintaining fast feedback cycles for developers and reliable validation for production deployments.

Unit Tests form the pyramid base and focus on individual component correctness. These tests run in milliseconds, use no external dependencies, and verify core business logic through deterministic inputs and outputs. Unit tests answer the question: "Does this component implement its contract correctly?" They use mocked dependencies, synthetic data, and focus on edge cases that might be rare in integration scenarios but could cause system failures.

Integration Tests form the pyramid middle and focus on component interaction correctness. These tests run in seconds, may use lightweight external dependencies (like embedded databases), and verify that components correctly communicate through their interfaces. Integration tests answer the question: "Do these components work together correctly?" They use realistic data flows, test error propagation between components, and verify that the system maintains consistency across component boundaries.

End-to-End Tests form the pyramid apex and focus on complete workflow correctness. These tests run in minutes, use production-like environments with real LLM APIs, and verify that the entire system delivers value to users. End-to-end tests answer the question: "Does the complete system solve the user's problem correctly?" They use real datasets, actual LLM providers, and validate the full evaluation lifecycle from dataset loading through final report generation.

Design Insight: The pyramid shape reflects both the number of tests at each level and their execution frequency. We need many fast unit tests running continuously, fewer integration tests running on commits, and even fewer end-to-end tests running before releases. This balance provides confidence without creating development friction.

Unit Test Organization

Unit tests focus on individual class and method correctness with comprehensive coverage of business logic, edge cases, and error conditions. Each component follows a consistent testing structure that makes tests readable, maintainable, and comprehensive.

Dataset Management Unit Tests verify core data structures and operations work correctly in isolation. These tests validate test case creation, dataset operations, version control logic, and data format conversions without touching the file system or external dependencies.

Test Category	Test Purpose	Example Test Cases
Test Case Validation	Verify <code>TestCase</code> schema compliance	Valid test case creation, missing required fields, invalid field types
Dataset Operations	Verify <code>Dataset</code> manipulation methods	Add test case, remove test case, dataset merging, duplicate handling
Version Control Logic	Verify versioning algorithms	Content hash computation, diff generation, merge conflict detection
Data Format Conversion	Verify import/export correctness	CSV to <code>TestCase</code> conversion, JSON schema validation, malformed data handling

The dataset loader unit tests use synthetic data that covers edge cases rarely seen in real datasets but critical for robustness. For example, testing CSV files with embedded commas, JSON with nested quotes, and malformed Unicode characters. These tests verify the loader's resilience without requiring actual external files.

Metrics Engine Unit Tests verify scoring algorithms produce correct results across the full range of inputs. These tests are particularly critical because metric correctness directly impacts evaluation quality, and bugs in scoring logic can invalidate entire evaluation runs.

Test Category	Test Purpose	Example Test Cases
Exact Match Logic	Verify string comparison accuracy	Identical strings, whitespace differences, case sensitivity, Unicode normalization
Fuzzy Match Logic	Verify normalization pipeline	Character normalization, punctuation removal, stop word handling, custom patterns
Semantic Similarity	Verify embedding-based scoring	Synonymous phrases, contradictory statements, embedding cache behavior, threshold calibration
LLM-as-Judge	Verify judge prompt generation	Prompt template rendering, response parsing, error handling, consistency controls
Score Aggregation	Verify statistical computations	Mean/median calculation, confidence intervals, outlier detection, weighted averages

Metrics unit tests use carefully crafted test cases with known expected outputs. For semantic similarity, this includes word pairs with established similarity ratings from linguistic datasets. For LLM-as-judge, this includes mock responses with various score formats to verify parsing robustness.

Evaluation Runner Unit Tests verify execution orchestration logic without actually calling LLM APIs or performing expensive operations. These tests focus on batch processing logic, concurrency control, caching behavior, and error handling using mock implementations.

Test Category	Test Purpose	Example Test Cases
Batch Processing	Verify batching algorithms	Batch size adaptation, load balancing, completion tracking, partial batch handling
Concurrency Control	Verify thread safety	Rate limiting accuracy, resource pool management, deadlock prevention, fair scheduling
Caching Logic	Verify cache behavior	Cache hit/miss logic, TTL expiration, cache eviction, content-based keying
Progress Tracking	Verify progress calculations	ETA computation, throughput measurement, failure rate tracking, statistical smoothing
Checkpoint/Recovery	Verify state persistence	Incremental checkpointing, WAL integrity, state reconstruction, corruption detection

Reporting Unit Tests verify statistical computations and report generation logic using synthetic evaluation results with known statistical properties. These tests ensure mathematical correctness and validate that insights are statistically sound.

Test Category	Test Purpose	Example Test Cases
Statistical Analysis	Verify computation accuracy	Mean/variance calculation, confidence intervals, significance testing, effect size computation
Regression Detection	Verify change detection	Performance degradation detection, statistical significance, false positive rates, baseline management
Failure Analysis	Verify clustering logic	Similarity-based grouping, representative selection, root cause hypotheses, improvement recommendations
Report Generation	Verify output formatting	HTML generation, chart rendering, PDF export, template processing

Integration Test Organization

Integration tests verify component interactions work correctly by testing realistic data flows through multiple components simultaneously. These tests use lightweight infrastructure (in-memory databases, mock HTTP servers) to simulate external dependencies while maintaining fast execution.

Cross-Component Data Flow Tests verify that data structures pass correctly between components and that interface contracts are properly implemented. These tests catch mismatches between what one component produces and what another component expects.

Integration Scenario	Components Tested	Validation Focus
Dataset to Metrics	<code>DatasetLoader</code> + <code>MetricRegistry</code>	Test case schema compatibility, metadata preservation, error propagation
Metrics to Runner	<code>MetricRegistry</code> + <code>BatchProcessor</code>	Metric configuration, result aggregation, concurrent execution
Runner to Analysis	<code>BatchProcessor</code> + <code>ScoreAggregator</code>	Result format consistency, statistical input validation, progress tracking
Analysis to Reporting	<code>ScoreAggregator</code> + <code>ReportGenerator</code>	Statistical output format, visualization data preparation, template compatibility

End-to-End Component Chains test complete workflows through multiple components to verify the system maintains data consistency and error handling across component boundaries. These tests use realistic datasets and configurations while mocking expensive external calls.

Error Propagation Tests verify that errors are correctly detected, contextualized, and propagated through the component stack. These tests simulate various failure scenarios and verify that error information is preserved and actionable.

Error Scenario	Component Chain	Expected Behavior
Invalid Dataset Format	<code>DatasetLoader</code> → <code>MetricRegistry</code>	Validation error with specific field information, graceful degradation
Metric Computation Failure	<code>MetricRegistry</code> → <code>BatchProcessor</code>	Error contextualization with test case ID, partial result handling
API Rate Limiting	<code>BatchProcessor</code> → <code>ProgressTracker</code>	Backoff behavior, progress preservation, automatic retry
Statistical Computation Error	<code>ScoreAggregator</code> → <code>ReportGenerator</code>	Insufficient data detection, alternative analysis methods, user guidance

Configuration Integration Tests verify that configuration changes propagate correctly through all components and that invalid configurations are detected early with clear error messages.

End-to-End Test Organization

End-to-end tests validate complete evaluation workflows using real external dependencies and production-like configurations. These tests run against actual LLM APIs, real datasets, and verify that the system produces valuable insights for practitioners.

Complete Evaluation Workflows test the full evaluation lifecycle from dataset loading through final report generation. These tests use representative datasets and verify that all system components work together to produce consistent, actionable results.

Workflow Scenario	Dataset Type	Evaluation Focus	Success Criteria
Small Dataset Evaluation	50 test cases, mixed difficulty	Correctness and completeness	All test cases processed, all metrics computed, statistical summary generated
Large Dataset Evaluation	1000+ test cases	Performance and reliability	Completion within time limits, resource usage within bounds, accurate progress tracking
Multi-Metric Evaluation	Exact match + semantic similarity + LLM-as-judge	Metric orchestration	All metrics applied appropriately, scores properly aggregated, no metric interference
Regression Detection	Current vs. baseline results	Change detection accuracy	Significant changes detected, false positives minimized, actionable recommendations

Production Readiness Tests verify the system performs correctly under production-like conditions including realistic datasets, actual LLM API latency and failure rates, and resource constraints.

User Acceptance Tests verify that the system produces insights that are actually valuable for LLM application developers. These tests use real-world datasets from various domains and verify that evaluation results help identify genuine model strengths and weaknesses.

Milestone Checkpoints

Each development milestone includes specific verification criteria that validate both functional correctness and system integration. These checkpoints provide clear success indicators and help identify issues early in development.

Milestone 1: Dataset Management Checkpoints

Dataset Loading Verification ensures the dataset management system correctly handles various input formats and produces consistent internal representations.

Checkpoint	Verification Method	Success Criteria	Common Issues
CSV Import	Load sample datasets from various domains	All test cases parsed correctly, schema validation passes, metadata preserved	Column mapping failures, encoding issues, malformed CSV handling
JSON Schema Validation	Test with valid and invalid JSON datasets	Valid datasets accepted, invalid datasets rejected with specific error messages	Schema drift, nested object handling, required field validation
Version Control	Create, modify, and version datasets	Version history accurate, diffs computed correctly, rollback functionality works	Content hash collisions, merge conflicts, version graph corruption
Dataset Splitting	Split datasets with various stratification strategies	Train/test/validation splits maintain proportions, no data leakage between splits	Random seed inconsistency, stratification imbalance, edge case handling

Data Integrity Validation verifies that dataset operations preserve data consistency and detect corruption early.

```
Test Command: python -m pytest tests/dataset_management/ -v --cov=dataset_management
Expected Output:
- All CSV, JSON, JSONL format tests pass
- Schema validation catches all malformed inputs
- Version control operations complete without corruption
- Dataset splits maintain statistical properties
```

The milestone 1 checkpoint should demonstrate loading a real evaluation dataset (such as a subset of GLUE or SuperGLUE), creating multiple versions with different test cases, and successfully splitting the data while preserving label distributions.

Milestone 2: Evaluation Metrics Checkpoints

Metric Correctness Verification ensures all scoring algorithms produce mathematically correct results across diverse inputs.

Metric Type	Verification Method	Success Criteria	Common Issues
Exact Match	Test with identical, similar, and different strings	Perfect precision/recall on known test cases, proper normalization handling	Case sensitivity bugs, whitespace handling, Unicode normalization
Fuzzy Match	Test with various string similarity levels	BLEU/ROUGE scores match reference implementations, edge cases handled correctly	N-gram extraction errors, smoothing parameter issues, empty string handling
Semantic Similarity	Test with synonym pairs and contrasting pairs	Similarity scores correlate with human judgments, threshold calibration accurate	Embedding model inconsistency, cache invalidation, similarity score interpretation
LLM-as-Judge	Test with mock judge responses and real API calls	Score parsing robust, consistency controls effective, error handling graceful	Prompt template bugs, response parsing failures, API rate limiting issues

Metric Integration Verification tests the metrics engine's ability to orchestrate multiple metrics and produce coherent aggregate scores.

```
Test Command: python -m pytest tests/metrics_engine/ -v --timeout=30
Expected Output:
- All individual metric tests pass with >95% accuracy on known benchmarks
- Multi-metric orchestration produces consistent results
- Custom metric registration and execution works correctly
- Metric applicability rules applied correctly
```

The milestone 2 checkpoint should demonstrate evaluating a set of model outputs using all metric types simultaneously, with results that align with expected behavior (e.g., identical outputs score 1.0 on exact match, synonymous outputs score high on semantic similarity).

Milestone 3: Evaluation Runner Checkpoints

Execution Performance Verification ensures the evaluation runner achieves target throughput and resource utilization while maintaining correctness.

Performance Aspect	Verification Method	Success Criteria	Common Issues
Batch Processing	Run evaluation with various batch sizes	Optimal batch size determined automatically, throughput targets met	Memory leaks, batch size oscillation, deadlock conditions
Concurrency Control	Test with high concurrency limits	Rate limiting respected, no race conditions, fair resource allocation	Thread pool exhaustion, rate limiter inaccuracy, resource contention
Result Caching	Run same evaluation multiple times	Cache hit rate >90% on repeated runs, cache consistency maintained	Cache key collisions, TTL issues, disk space management
Progress Tracking	Monitor long-running evaluations	ETA accuracy within 10%, progress updates responsive, failure tracking accurate	Progress calculation errors, UI freezing, incomplete failure reporting

Reliability Verification tests the system's ability to handle failures gracefully and resume operations correctly.

```
Test Command: python -m pytest tests/evaluation_runner/ -v --integration
Expected Behavior:
- Evaluation completes successfully with 100+ test cases
- Progress tracking shows steady advancement with accurate ETA
- Cache reduces API calls by >90% on repeated runs
- System recovers correctly from simulated API failures
```

The milestone 3 checkpoint should demonstrate running a complete evaluation on a moderately-sized dataset (200+ test cases) while simulating various failure conditions and verifying that the system recovers gracefully with minimal progress loss.

Milestone 4: Reporting & Analysis Checkpoints

Statistical Analysis Verification ensures all mathematical computations are correct and that insights are statistically valid.

Analysis Component	Verification Method	Success Criteria	Common Issues
Score Aggregation	Test with datasets of known statistical properties	Mean/median/variance calculations accurate to 4 decimal places, confidence intervals valid	Floating point precision issues, outlier handling, small sample biases
Regression Detection	Compare identical and different result sets	True regressions detected with $p < 0.05$, false positive rate <5%	Multiple comparison issues, insufficient sample sizes, effect size interpretation
Failure Analysis	Test with datasets containing known failure patterns	Similar failures clustered correctly, root causes identified accurately	Clustering parameter sensitivity, similarity metric choice, interpretation quality
Report Generation	Generate reports with various dataset sizes	Reports complete within time limits, visualizations render correctly, exports succeed	Template rendering failures, chart generation errors, PDF export issues

Insight Quality Verification tests that generated reports contain actionable insights that help users improve their LLM applications.

```
Test Command: python -m pytest tests/reporting_analysis/ -v --slow
Expected Output:
- Statistical computations pass accuracy benchmarks
- Regression detection achieves target sensitivity/specificity
- Failure clustering produces interpretable groups
- Reports generate successfully with interactive visualizations
```

The milestone 4 checkpoint should demonstrate generating a comprehensive evaluation report that identifies genuine performance differences between two model configurations and provides specific, actionable recommendations for improvement.

Cross-Milestone Integration Verification ensures all components work together correctly to deliver the complete evaluation workflow.

```
Full System Test Command:
python -m pytest tests/end_to_end/ -v --timeout=300 --real-apis

Expected End-to-End Behavior:
1. Load multi-format dataset (CSV + JSON) with 500+ diverse test cases
2. Execute evaluation using multiple metrics (exact + semantic + LLM-judge)
3. Complete evaluation within resource limits (memory <2GB, time <10min)
4. Generate comprehensive report with statistical analysis and visualizations
5. Detect meaningful differences when comparing against baseline results
6. Export results in multiple formats (HTML, PDF, JSON) successfully
```

The final checkpoint validates that a practitioner can load their evaluation dataset, configure appropriate metrics, run a complete evaluation, and receive actionable insights for improving their LLM application - all within reasonable time and resource constraints.

Implementation Guidance

The testing strategy implementation focuses on creating a robust, maintainable test suite that provides fast feedback during development while ensuring production reliability.

Technology Recommendations

Testing Component	Simple Option	Advanced Option
Unit Testing Framework	pytest with standard assertions	pytest + hypothesis for property-based testing
Mock/Stub Framework	unittest.mock (built-in)	pytest-mock with advanced fixtures
Integration Testing	pytest with temporary files	testcontainers for realistic dependencies
End-to-End Testing	pytest with real API calls	pytest-xdist for parallel execution
Test Data Management	JSON fixtures in test files	factory_boy for dynamic test data generation
Coverage Analysis	coverage.py with pytest-cov	coveragepy with branch coverage and reporting
Performance Testing	pytest-benchmark for timing	locust for load testing evaluation workflows

Recommended File Structure

The test organization mirrors the main codebase structure while providing clear separation between test types and shared testing utilities.

```

llm-evaluation-framework/
├── tests/
│   ├── unit/                      # Fast, isolated component tests
│   │   ├── dataset_management/
│   │   │   ├── test_dataset_loader.py
│   │   │   ├── test_dataset_version_control.py
│   │   │   └── test_dataset_splitting.py
│   │   ├── metrics_engine/
│   │   │   ├── test_exact_match_metric.py
│   │   │   ├── test_semantic_similarity_metric.py
│   │   │   ├── test_llm_judge_metric.py
│   │   │   └── test_metric_registry.py
│   │   ├── evaluation_runner/
│   │   │   ├── test_batch_processor.py
│   │   │   ├── test_progress_tracker.py
│   │   │   └── test_result_cache.py
│   │   └── reporting_analysis/
│   │       ├── test_score_aggregator.py
│   │       ├── test_regression_detector.py
│   │       └── test_report_generator.py
│   ├── integration/               # Cross-component interaction tests
│   │   ├── test_dataset_to_metrics.py
│   │   ├── test_metrics_to_runner.py
│   │   ├── test_runner_to_analysis.py
│   │   └── test_error_propagation.py
│   ├── end_to_end/                # Complete workflow tests
│   │   ├── test_full_evaluation_workflow.py
│   │   ├── test_production_scenarios.py
│   │   └── test_user_acceptance.py
│   ├── fixtures/                 # Shared test data
│   │   ├── datasets/
│   │   │   ├── sample_dataset.csv
│   │   │   ├── sample_dataset.json
│   │   │   └── malformed_dataset.csv
│   │   ├── evaluation_results/
│   │   │   ├── baseline_results.json
│   │   │   └── regression_results.json
│   │   └── mock_responses/
│   │       ├── llm_api_responses.json
│   │       └── embedding_api_responses.json
│   ├── utils/                     # Testing utilities and helpers
│   │   ├── mock_providers.py      # Mock LLM and embedding providers
│   │   ├── test_data_factory.py   # Dynamic test data generation
│   │   ├── assertion_helpers.py  # Custom assertion functions
│   │   └── performance_utils.py # Performance testing utilities
│   └── conftest.py                # Pytest configuration and shared fixtures
└── pytest.ini                   # Pytest configuration file
└── coverage.ini                 # Coverage analysis configuration

```

Infrastructure Starter Code

Mock Provider Implementation - Complete mock implementation for testing without external API dependencies:

```
"""Mock providers for testing LLM and embedding APIs without external dependencies."""
```

PYTHON

```
import asyncio
import json
import random
from pathlib import Path
from typing import Dict, List, Optional, Any
from unittest.mock import Mock, AsyncMock
from llm_evaluation.providers.base import LLMPromoter, EmbeddingProvider
from llm_evaluation.data_model import LLMPromoterConfig

class MockLLMPromoter(LLMPromoter):
    """Mock LLM provider that returns deterministic responses for testing."""

    def __init__(self, config: LLMPromoterConfig, response_file: Optional[Path] = None):
        super().__init__(config)
        self.call_count = 0
        self.response_latency = 0.1 # Simulated API latency
        self.failure_rate = 0.0 # Probability of API failure

        # Load pre-defined responses for deterministic testing
        self.responses = {}
        if response_file and response_file.exists():
            self.responses = json.loads(response_file.read_text())

    async def generate_response(self, prompt: str, **kwargs) -> str:
        """Generate mock response with configurable behavior."""
        await asyncio.sleep(self.response_latency) # Simulate API latency

        self.call_count += 1
```

```
# Simulate API failures for reliability testing

if random.random() < self.failure_rate:
    raise Exception(f"Mock API failure (call #{self.call_count})")

# Return pre-defined response if available

prompt_hash = str(hash(prompt))

if prompt_hash in self.responses:
    return self.responses[prompt_hash]

# Generate deterministic response based on prompt hash

response_variants = [
    "This is a test response.",
    "Mock LLM generated this output.",
    "Simulated response for testing purposes.",
    "Deterministic test output based on input hash."
]

return response_variants[abs(hash(prompt)) % len(response_variants)]


class MockEmbeddingProvider(EmbeddingProvider):

    """Mock embedding provider that returns consistent embeddings for testing."""

    def __init__(self, dimension: int = 384):
        self.dimension = dimension
        self.call_count = 0
        self.embedding_cache = {}

    async def get_embeddings(self, texts: List[str], model: str = "mock") -> List[List[float]]:
        """Generate consistent embeddings based on text hash."""
```

```
    self.call_count += 1

    embeddings = []

    for text in texts:

        if text in self.embedding_cache:

            embeddings.append(self.embedding_cache[text])

        else:

            # Generate deterministic embedding based on text hash

            random.seed(hash(text))

            embedding = [random.gauss(0, 1) for _ in range(self.dimension)]

            # Normalize to unit length for cosine similarity

            magnitude = sum(x**2 for x in embedding) ** 0.5

            embedding = [x / magnitude for x in embedding]

            self.embedding_cache[text] = embedding

            embeddings.append(embedding)

    return embeddings


def create_mock_evaluation_config() -> Dict[str, Any]:

    """Create realistic evaluation configuration for testing."""

    return {

        "dataset_storage_path": Path("/tmp/test_datasets"),

        "max_concurrent_evaluations": 4,

        "llm_providers": {

            "primary": LLMProviderConfig(

                provider="mock",

                api_key="test-key",

                model="mock-model",
```

```
    max_tokens=1000,  
    temperature=0.0  
)  
,  
"cache_config": {  
    "cache_dir": Path("/tmp/test_cache"),  
    "default_ttl": 3600  
,  
    "rate_limits": {  
        "requests_per_minute": 60,  
        "burst_limit": 10  
    }  
}
```

Test Data Factory - Dynamic test data generation for comprehensive testing:

```
"""Factory functions for generating test data with realistic characteristics."""

import random

from datetime import datetime, timedelta

from typing import List, Dict, Any, Optional

from faker import Faker

from llm_evaluation.data_model import TestCase, Dataset, EvaluationResult, MetricResult

fake = Faker()

Faker.seed(42) # Deterministic test data

class TestDataFactory:

    """Factory for generating realistic test data for evaluation framework testing."""

    @staticmethod
    def create_test_case(
        case_id: Optional[str] = None,
        prompt: Optional[str] = None,
        expected_output: Optional[str] = None,
        tags: Optional[List[str]] = None,
        difficulty: str = "medium"
    ) -> TestCase:
        """Create a realistic test case with optional customization."""

        return TestCase(
            case_id=case_id or fake.uuid4(),
            prompt=prompt or fake.paragraph(nb_sentences=3),
            expected_output=expected_output or fake.paragraph(nb_sentences=2),
            tags=tags or random.sample(["qa", "summarization", "classification", "generation"], 2),
            difficulty=difficulty,
            metadata={}
```

```
        "domain": random.choice(["medical", "legal", "technical", "general"]),

        "length": random.choice(["short", "medium", "long"]),

        "created_by": fake.name()

    }

)

@staticmethod

def create_dataset(

    name: Optional[str] = None,

    test_cases: Optional[List[TestCase]] = None,

    num_cases: int = 50

) -> Dataset:

    """Create a realistic dataset with specified number of test cases."""

    if test_cases is None:

        test_cases = [

            TestDataFactory.create_test_case()

            for _ in range(num_cases)

        ]

    return Dataset(

        name=name or f"test_dataset_{fake.uuid4()[:8]}",

        test_cases=test_cases,

        version="1.0.0",

        description=fake.sentence(),

        created_at=datetime.now()

    )



@staticmethod

def create_metric_result(
```

```

        score: Optional[float] = None,
        explanation: Optional[str] = None,
        with_error: bool = False
    ) -> MetricResult:
        """Create realistic metric result with optional error simulation."""
        if with_error:
            return MetricResult(
                score=0.0,
                metadata={},
                explanation=None,
                computation_time=random.uniform(0.1, 0.5),
                error_info="Simulated computation error for testing",
                preprocessing_applied=[]
            )
        return MetricResult(
            score=score if score is not None else random.uniform(0.0, 1.0),
            metadata={"confidence": random.uniform(0.8, 1.0)},
            explanation=explanation or fake.sentence(),
            computation_time=random.uniform(0.01, 0.1),
            error_info=None,
            preprocessing_applied=random.sample(["normalize_whitespace", "normalize_case"], 1)
        )
    )

```

Core Testing Skeleton Code

Unit Test Template - Structured template for component unit tests:

```
"""Template for unit tests following framework testing patterns."""

import pytest

from unittest.mock import Mock, patch, MagicMock

from pathlib import Path

from llm_evaluation.dataset_management.dataset_loader import DatasetLoader

from llm_evaluation.data_model import TestCase, Dataset

from tests.utils.test_data_factory import TestDataFactory


class TestDatasetLoader:

    """Unit tests for DatasetLoader component."""

    @pytest.fixture
    def loader(self):
        """Create DatasetLoader instance for testing."""
        return DatasetLoader()

    @pytest.fixture
    def sample_csv_data(self):
        """Create sample CSV data for testing."""
        # TODO 1: Create CSV string with proper headers (prompt, expected_output, tags, difficulty)
        # TODO 2: Include edge cases (embedded commas, quotes, unicode characters)
        # TODO 3: Add malformed rows for error handling testing
        # Hint: Use realistic data that tests column mapping logic
        pass

    def test_load_csv_dataset_success(self, loader, sample_csv_data, tmp_path):
        """Test successful CSV dataset loading."""
        # TODO 1: Write sample_csv_data to temporary file
        # TODO 2: Call loader.load_dataset() with CSV file path
```

```

# TODO 3: Verify returned Dataset has correct number of TestCase objects

# TODO 4: Verify TestCase fields mapped correctly from CSV columns

# TODO 5: Verify metadata preserved and tags parsed correctly

pass


def test_load_csv_dataset_malformed_file(self, loader, tmp_path):
    """Test CSV loading with malformed data."""

    # TODO 1: Create CSV file with missing required columns

    # TODO 2: Verify DatasetLoader raises appropriate validation exception

    # TODO 3: Verify error message identifies specific missing columns

    # TODO 4: Test with various malformed scenarios (empty file, wrong encoding, etc.)

    pass


def test_automatic_column_mapping(self, loader, tmp_path):
    """Test automatic mapping of CSV columns to TestCase schema."""

    # TODO 1: Create CSV with non-standard column names (question, answer, category)

    # TODO 2: Verify loader correctly maps to standard schema (prompt, expected_output, tags)

    # TODO 3: Test mapping confidence scores and fallback behavior

    # TODO 4: Verify unmappable columns handled gracefully

    pass


@patch('llm_evaluation.dataset_management.dataset_loader.json.load')

def test_json_loading_with_mocked_file_io(self, mock_json_load, loader):
    """Test JSON loading logic with mocked file operations."""

    # TODO 1: Configure mock_json_load to return test data structure

    # TODO 2: Call loader.load_dataset() with JSON file path

    # TODO 3: Verify JSON parsing called with correct parameters

    # TODO 4: Verify TestCase objects created from JSON structure

    # TODO 5: Test error handling when JSON structure invalid

```

pass

Milestone Checkpoint Implementation

Milestone Validation Script - Automated verification of milestone completion:

```
"""Automated validation script for milestone checkpoints."""

import subprocess

import json

import tempfile

from pathlib import Path

from typing import List, Dict, Any

from llm_evaluation.data_model import Dataset, TestCase

from llm_evaluation.dataset_management import DatasetLoader

from tests.utils.test_data_factory import TestDataFactory


class MilestoneValidator:

    """Validates milestone completion with automated checks."""

    def __init__(self, project_root: Path):

        self.project_root = project_root

        self.validation_results = {}



    def validate_milestone_1_dataset_management(self) -> Dict[str, bool]:

        """Validate Milestone 1: Dataset Management completion."""

        results = {}



        # TODO 1: Test CSV import functionality

        # Create sample CSV file with 100 test cases

        # Load using DatasetLoader and verify all cases imported correctly

        # Check schema validation passes and metadata preserved



        # TODO 2: Test dataset versioning

        # Create initial dataset version

        # Modify dataset and create new version
```

```
# Verify version diff computed correctly and rollback works

# TODO 3: Test dataset splitting

# Split dataset into train/test/validation with 70/20/10 ratio

# Verify no data leakage between splits

# Check stratification maintains tag distribution

# TODO 4: Test golden examples functionality

# Mark subset of test cases as golden examples

# Verify golden examples preserved across operations

# Check golden example retrieval and filtering

return results

def validate_milestone_2_metrics_engine(self) -> Dict[str, bool]:
    """Validate Milestone 2: Evaluation Metrics completion."""
    results = {}

    # TODO 1: Test exact match metric accuracy

    # Create test cases with known exact/non-exact matches

    # Verify exact match metric returns 1.0 for identical strings

    # Test fuzzy matching with various normalization levels

    # TODO 2: Test semantic similarity metric

    # Use synonym pairs with known similarity ratings

    # Verify semantic similarity scores correlate with expected ratings

    # Test embedding caching and threshold calibration

    # TODO 3: Test LLM-as-judge functionality
```

```
# Mock LLM responses with various score formats

# Verify score parsing handles different response styles

# Test consistency controls and error handling

# TODO 4: Test custom metric registration

# Register custom metric plugin

# Verify metric discovered and executed correctly

# Test metric applicability rules and score normalization

return results

def run_all_validations(self) -> Dict[str, Dict[str, bool]]:
    """Run all milestone validations and return comprehensive results."""

    # TODO 1: Execute each milestone validation

    # TODO 2: Compile results into structured report

    # TODO 3: Generate validation summary with pass/fail status

    # TODO 4: Create detailed error report for failed validations

    # TODO 5: Return results in format suitable for CI/CD integration

    pass

def main():
    """Main validation entry point for CI/CD integration."""

    # TODO 1: Parse command line arguments for milestone selection

    # TODO 2: Initialize MilestoneValidator with project root

    # TODO 3: Run specified milestone validations

    # TODO 4: Print results in CI-friendly format

    # TODO 5: Exit with appropriate status code (0 for success, 1 for failures)

    pass
```

```
if __name__ == "__main__":
    main()
```

Debugging Guide

Milestone(s): All milestones (1-4) - Debugging techniques are essential throughout dataset management (Milestone 1), metrics computation (Milestone 2), evaluation execution (Milestone 3), and reporting (Milestone 4)

Think of debugging an LLM evaluation framework like being a detective investigating a crime scene. You have symptoms (the evaluation failed or produced unexpected results), potential suspects (various components that could be malfunctioning), and evidence (logs, cached data, intermediate results). The key is knowing where to look first, what clues to gather, and how to systematically eliminate possibilities until you find the root cause.

The debugging challenge is particularly complex in evaluation systems because failures can cascade across multiple layers. A dataset loading error might not surface until metric computation, an embedding cache corruption might cause semantic similarity scores to drift silently, and API rate limiting might create intermittent failures that are difficult to reproduce. Understanding these interaction patterns is crucial for effective debugging.

Common Implementation Bugs

The most frequent bugs developers encounter when building and using the evaluation framework fall into predictable patterns. Understanding these patterns helps developers quickly identify and fix issues without extensive investigation.

Dataset Management Bugs

Schema Validation Failures

The most common dataset management bug involves schema validation failures that aren't immediately obvious. When the `DatasetLoader` attempts to map CSV columns to the `TestCase` schema, subtle mismatches can cause entire datasets to be rejected or partially loaded with corrupted data.

Bug Pattern	Symptom	Root Cause	Fix
Case sensitivity mismatch	"prompt" column not found error	CSV has "Prompt" but loader expects "prompt"	Add case-insensitive column mapping
Extra whitespace in headers	Random columns marked as missing	CSV headers have trailing spaces	Strip whitespace in column name normalization
Unicode encoding issues	Garbled text in loaded test cases	CSV saved with different encoding than UTF-8	Detect encoding automatically or force UTF-8
Missing required fields	Schema validation error after load	Some test cases lack <code>expected_output</code>	Add validation during load with clear error messages
Tag parsing failures	Tags appear as strings instead of lists	JSON tags serialized as strings in CSV	Parse tag columns as JSON arrays with fallback

⚠️ Pitfall: Silent Schema Violations Many developers assume that if `load_dataset()` returns successfully, all test cases are valid. However, the loader might skip invalid rows silently or apply default values that mask data quality issues. Always check the returned `Dataset.test_cases` count against the expected number of rows and validate a sample of loaded cases manually.

Version Control Conflicts

Dataset versioning introduces merge conflicts that can corrupt the version history if not handled properly. The `DatasetVersionControl` system tracks changes using content hashes, but concurrent modifications can create inconsistent states.

Bug Pattern	Symptom	Root Cause	Fix
Hash collision detection failure	Different datasets with same version ID	Weak hashing algorithm or insufficient content coverage	Use SHA-256 with complete test case serialization
Merge conflict resolution	Lost changes after version merge	Automatic merge overwrites manual edits	Implement three-way merge with conflict markers
Circular version references	Infinite loop during version traversal	Parent-child relationships create cycles	Validate version graph for cycles during creation
Concurrent write corruption	Version metadata inconsistent	Multiple processes writing simultaneously	Use file locking or atomic rename operations
Orphaned version cleanup	Disk space grows without bound	Old versions never garbage collected	Implement reference counting for version cleanup

⚠️ Pitfall: Version Drift When multiple team members work on the same dataset, their local versions can diverge silently. The framework tracks content hashes, but developers often don't notice that their "latest" version differs from others. Always compare version IDs and change summaries before assuming you're working with the same dataset version.

Metrics Engine Bugs

Embedding Cache Corruption

The `EmbeddingCache` stores computed embeddings to avoid redundant API calls, but cache corruption can cause semantic similarity scores to become unreliable. This is particularly insidious because the corruption often affects only a subset of embeddings, making the problem appear intermittent.

Bug Pattern	Symptom	Root Cause	Fix
Stale embedding retrieval	Semantic similarity scores don't match expected values	Cache returns old embeddings after model update	Include model version in cache key
Partial cache corruption	Some similarity scores are obviously wrong (negative, >1.0)	Database corruption affects specific entries	Validate cached embeddings before use
Memory pressure eviction	Embeddings cache hit rate drops unexpectedly	System memory pressure causes cache eviction	Monitor memory usage and implement cache size limits
Concurrent access corruption	Random cache misses despite recent computation	Multiple processes writing to cache simultaneously	Use database transactions or file locking
Text preprocessing inconsistency	Same text produces different embeddings	Text preprocessed differently during cache storage vs retrieval	Store preprocessing metadata with cached embeddings

⚠ Pitfall: Silent Cache Poisoning When the embedding model changes (e.g., upgrading from text-embedding-ada-002 to text-embedding-3-small), cached embeddings from the old model remain valid according to the cache logic but are semantically incompatible. This causes semantic similarity scores to drift gradually as cached and fresh embeddings get mixed. Always invalidate the cache when changing embedding models.

LLM-as-Judge Inconsistency

The LLM-as-judge evaluation pattern is notoriously unreliable because language models can produce different scores for identical inputs. This non-determinism makes debugging particularly challenging since the same test might pass or fail on different runs.

Bug Pattern	Symptom	Root Cause	Fix
Temperature sensitivity	Judge scores vary widely between runs	Non-zero temperature causes random sampling	Set temperature=0.0 for deterministic output
Prompt engineering failures	Judge produces unparseable scores ("Good" instead of numbers)	Judge prompt doesn't constrain output format sufficiently	Add explicit format requirements and examples
Context length overflow	Judge truncates long inputs silently	Model response or expected output exceeds context window	Implement chunking or fallback to exact match
Model-specific biases	Judge consistently favors certain response patterns	Different judge models have different scoring tendencies	Calibrate score thresholds per judge model
Parsing edge cases	Score extraction fails on valid judge responses	Regex or JSON parsing doesn't handle all output formats	Use robust parsing with multiple fallback strategies

⚠ Pitfall: Judge Model Drift Cloud LLM providers occasionally update their models behind the same API endpoint, causing judge scoring to drift over time even with identical prompts and temperature settings. This breaks evaluation reproducibility. Always log the exact model version used for judging and monitor score distributions for unexpected changes.

Evaluation Runner Bugs

Concurrency and Rate Limiting Issues

The `EvaluationRunner` processes test cases in parallel to maximize throughput, but this introduces race conditions and rate limiting complications that can cause evaluations to fail or produce incorrect results.

Bug Pattern	Symptom	Root Cause	Fix
Rate limit cascade failures	Evaluation fails completely after initial rate limit hit	No exponential backoff, all retries happen simultaneously	Implement jittered exponential backoff with per-provider limits
Cache race conditions	Same prompt evaluated multiple times simultaneously	Multiple workers check cache, find miss, make same API call	Use atomic cache operations or request deduplication
Memory exhaustion	Evaluation slows dramatically or crashes with OOM	Large result sets stored in memory without streaming	Implement result streaming and periodic memory cleanup
Progress tracking corruption	Progress bar shows >100% or goes backwards	Concurrent updates to progress counters without synchronization	Use atomic operations or message passing for progress updates
Checkpoint corruption	Recovery loads partial or inconsistent state	Checkpoint written while evaluation state is changing	Use copy-on-write or stop-the-world checkpointing

⚠ Pitfall: Thundering Herd on Recovery When an evaluation resumes from a checkpoint, all pending test cases often start processing simultaneously, overwhelming the LLM API and triggering rate limits immediately. This can make recovery slower than restarting from scratch. Implement gradual ramp-up after recovery, starting with small batch sizes and increasing as the system proves stable.

Result Caching Edge Cases

The `PromptCache` stores LLM responses to avoid redundant API calls, but cache key generation and invalidation logic contains subtle bugs that can cause stale results to be served or cache misses where hits should occur.

Bug Pattern	Symptom	Root Cause	Fix
Hash collision false positives	Different prompts return identical cached responses	Weak hash function causes different prompts to map to same key	Use cryptographic hash (SHA-256) of full prompt content
Provider config ignored in key	Same prompt with different models/temperatures returns same cached result	Cache key doesn't include all provider configuration parameters	Include full <code>LLMProviderConfig</code> in cache key generation
TTL inconsistency	Cache returns stale results after configured expiration	System clock drift or timezone issues affect TTL calculation	Use UTC timestamps and validate system clock synchronization
Cache key encoding issues	Unicode prompts produce inconsistent cache keys	String encoding differences affect hash computation	Normalize all text to UTF-8 before hashing
Large prompt truncation	Cache misses for prompts that should hit	Hash computation truncates very long prompts	Use streaming hash computation for large inputs

⚠ Pitfall: Development vs Production Cache Differences Developers often test with small datasets and see excellent cache hit rates, but production evaluations with larger, more diverse datasets experience much lower hit rates. This happens because development datasets often contain duplicate or very similar prompts, while production datasets are more varied. Always test caching performance with production-scale, production-diversity datasets.

Reporting and Analysis Bugs

Statistical Analysis Errors

The reporting system performs statistical analysis to detect regressions and generate insights, but statistical computations are prone to edge cases that can produce misleading or invalid results.

Bug Pattern	Symptom	Root Cause	Fix
Small sample size inflation	Statistical significance detected with tiny effect sizes	Insufficient sample size makes noise appear significant	Enforce minimum sample size requirements before statistical tests
Multiple comparison errors	False positive regression alerts	Testing many metrics simultaneously inflates Type I error rate	Apply Bonferroni or FDR correction for multiple comparisons
Outlier contamination	Mean scores drastically affected by few extreme values	No outlier detection before computing summary statistics	Use robust statistics (median, IQR) or outlier removal
Distribution assumption violations	t-tests applied to non-normal data	Statistical tests assume normality but scores are skewed	Use non-parametric tests or transform data to normality
Missing data handling	Analysis excludes failed test cases without adjustment	Metrics computed only on successful cases bias results upward	Use intent-to-treat analysis or explicitly model failure rates

⚠ Pitfall: Simpson's Paradox in Aggregation When aggregating scores across different test case categories, overall performance can appear to improve even when every individual category got worse, or vice versa. This happens when the mix of test case types changes between evaluations. Always report both overall and per-category statistics, and investigate any cases where trends diverge.

Visualization and Export Failures

Report generation involves complex data transformations and external dependencies that can fail in hard-to-diagnose ways, often producing partially corrupt outputs that appear valid at first glance.

Bug Pattern	Symptom	Root Cause	Fix
Chart rendering failures	Reports contain blank spaces where charts should appear	JavaScript errors in chart library or data format mismatches	Validate chart data schema and handle rendering exceptions
PDF export corruption	Generated PDFs are unreadable or truncated	HTML to PDF conversion fails on complex layouts	Simplify CSS, avoid unsupported features, test with headless browser
Unicode handling in exports	Special characters appear as boxes or question marks	Export pipeline doesn't preserve UTF-8 encoding	Ensure UTF-8 encoding throughout export pipeline
Large dataset memory issues	Report generation fails or takes extremely long	Visualization libraries load entire dataset into memory	Implement data sampling or streaming for large reports
Template variable injection	Report shows raw template syntax instead of values	Template rendering fails but error is silently ignored	Validate template variable binding and handle missing data

⚠ Pitfall: Browser-Specific Rendering Differences HTML reports often render correctly in one browser but break in others due to CSS compatibility issues or JavaScript differences. This is particularly problematic when stakeholders use

different browsers than developers. Always test report rendering in multiple browsers and use progressive enhancement for advanced features.

Debugging Techniques

Effective debugging of the evaluation framework requires systematic approaches tailored to the distributed, asynchronous nature of the system. The following techniques provide structured methodologies for diagnosing and resolving issues across all components.

Structured Diagnostic Approach

Symptom Classification and Triage

When an evaluation fails or produces unexpected results, the first step is classifying the symptom to determine which diagnostic path to follow. Different symptom types require different investigation strategies and have different urgency levels.

Symptom Category	Examples	Investigation Priority	Initial Diagnostic Steps
Complete evaluation failure	Process crashes, no results generated	Critical - investigate immediately	Check process logs, validate dataset loading, verify API credentials
Partial evaluation failure	Some test cases succeed, others fail	High - may indicate systematic issues	Analyze failure patterns by tags, difficulty, prompt length
Score distribution anomalies	Unexpectedly high/low scores, narrow distributions	Medium - may indicate metric bugs	Compare score histograms with baselines, validate metric implementations
Performance degradation	Evaluation takes much longer than expected	Medium - check for resource issues	Monitor CPU/memory usage, analyze batch processing efficiency
Report generation issues	Missing charts, formatting problems, export failures	Low - functional but presentation problems	Validate data pipeline inputs, check template rendering

Component Isolation Strategy

The evaluation framework consists of multiple interacting components, making it challenging to determine which component is responsible for a given issue. Systematic component isolation helps narrow down the investigation scope.

- 1. Start with the outermost component:** Begin investigation at the point where you first observe the problem (usually the `EvaluationCoordinator` or report output).
- 2. Trace the data flow backwards:** Follow the data transformation chain in reverse, validating inputs and outputs at each stage.
- 3. Use component boundaries as checkpoints:** Each major component (`DatasetLoader`, `MetricRegistry`, `BatchProcessor`, etc.) should provide validation methods to check its internal state.
- 4. Isolate external dependencies:** Test with mock implementations of LLM APIs, file systems, and databases to determine if the issue is internal or external.

5. **Reduce to minimal test case:** Create the smallest possible dataset and simplest possible configuration that reproduces the issue.

Log Analysis and Correlation

The framework generates logs from multiple components that execute concurrently. Effective log analysis requires correlating events across components and extracting meaningful patterns from high-volume log streams.

Log Analysis Technique	When to Use	Implementation
Correlation ID tracking	Tracing requests across components	Include <code>correlation_id</code> in all log messages for a single evaluation run
Timeline reconstruction	Understanding sequence of events leading to failure	Parse timestamps and reconstruct chronological event sequence
Error pattern matching	Identifying systematic vs random failures	Group errors by message pattern and analyze frequency distributions
Performance bottleneck analysis	Diagnosing slow evaluations	Extract timing information and identify longest-running operations
Resource utilization correlation	Connecting performance issues to resource constraints	Correlate timing logs with system resource metrics

Key Insight: The most valuable debugging information comes from the moments just before a failure occurs, not the failure itself. Configure logging to capture detailed state information during normal operation, not just error conditions.

Interactive Debugging Tools

Evaluation State Inspection

The framework provides programmatic interfaces for inspecting evaluation state at runtime, allowing developers to diagnose issues without restarting or modifying evaluations.

The `EvaluationCoordinator` exposes state inspection methods that can be called from a debugging interface or interactive Python session:

Inspection Method	Information Provided	Typical Use Cases
<code>get_active_evaluations()</code>	Currently running evaluations with progress stats	Checking if evaluation is stuck or making progress
<code>get_component_health()</code>	Status of all major components (dataset loader, metrics, etc.)	Identifying which component is failing
<code>get_cache_statistics()</code>	Hit rates, memory usage, error counts for all caches	Diagnosing cache-related performance issues
<code>get_batch_statistics()</code>	Current batch sizes, processing rates, error rates	Optimizing concurrency and identifying bottlenecks
<code>get_provider_health()</code>	API response times, error rates, rate limit status for each LLM provider	Diagnosing API-related failures

Interactive Metric Testing

Metric bugs are often subtle and require testing with specific inputs to reproduce. The framework provides an interactive metric testing interface that allows developers to evaluate individual metrics with custom inputs and inspect intermediate computation steps.

The metric testing workflow follows this pattern:

- 1. Load a specific test case:** Select a failing test case or create a synthetic case that demonstrates the issue.
- 2. Execute metric in isolation:** Run the problematic metric alone, bypassing the normal evaluation pipeline.
- 3. Inspect preprocessing steps:** Examine how text preprocessing affects the inputs before metric computation.
- 4. Trace computation steps:** For complex metrics like semantic similarity, inspect embedding generation and similarity calculation separately.
- 5. Compare with expected behavior:** Run the same inputs through alternative implementations or manual calculations.

Cache and Storage Debugging

Many evaluation issues stem from corrupted or inconsistent cached data. The framework provides utilities for inspecting and repairing cache contents without losing valuable cached results.

Debugging Utility	Purpose	Usage Scenario
Cache validation scanner	Detect corrupted cache entries	Run after system crashes or storage issues
Cache key collision detector	Find hash collisions causing incorrect cache hits	Investigate cases where different inputs return same results
Embedding consistency checker	Verify embedding cache matches current model	Run after model updates or provider changes
Dataset version integrity checker	Validate version history and detect corruption	Investigate version control issues
Checkpoint recovery validator	Verify checkpoint data can be loaded correctly	Test recovery capabilities without actually crashing

Systematic Testing and Validation

Regression Testing Framework

As the evaluation framework evolves, changes can introduce subtle bugs that only manifest under specific conditions. A comprehensive regression testing framework helps catch these issues before they affect production evaluations.

The regression testing strategy operates at multiple levels:

Unit-level regression tests validate individual components with synthetic data that exercises edge cases and error conditions. These tests run quickly and provide immediate feedback during development.

Integration-level regression tests use real datasets and LLM APIs to validate cross-component interactions. These tests take longer but catch issues that unit tests miss.

End-to-end regression tests run complete evaluations with known datasets and verify that results match historical baselines within acceptable tolerance ranges.

Performance regression tests measure evaluation throughput, memory usage, and API efficiency to detect performance degradations that might not affect correctness but impact user experience.

Critical Principle: Regression tests must be deterministic despite the non-deterministic nature of LLM APIs. Use fixed random seeds, cached responses, or mock providers to ensure test results are reproducible.

Baseline Management and Drift Detection

The evaluation framework's primary purpose is detecting changes in model performance over time. However, the framework itself can introduce measurement drift that masquerades as model performance changes. Systematic baseline management helps distinguish real performance changes from measurement artifacts.

Baseline establishment involves running the same evaluation multiple times with identical configurations to establish natural score variation ranges. This variation becomes the threshold for detecting real performance changes.

Drift detection monitors evaluation results for systematic changes that might indicate framework bugs rather than model changes. Examples include gradual score inflation due to cache corruption or sudden score shifts due to API provider model updates.

Control group evaluation runs a subset of test cases with a fixed, well-understood model (like GPT-3.5-turbo with fixed parameters) alongside production evaluations. Changes in control group scores indicate measurement issues rather than model performance changes.

Cross-validation with external tools periodically validates framework results against other evaluation tools or manual assessment to detect systematic biases or calibration drift.

Performance Profiling and Optimization

Resource Usage Monitoring

Evaluation performance issues often stem from resource bottlenecks that aren't immediately obvious. Systematic resource monitoring helps identify these bottlenecks and guide optimization efforts.

Resource Type	Monitoring Approach	Common Bottlenecks	Optimization Strategies
CPU utilization	Profile CPU usage during batch processing	JSON parsing, text preprocessing, statistical computations	Optimize data structures, use compiled libraries, implement caching
Memory consumption	Track memory growth during evaluation	Large datasets loaded entirely into memory, embedding cache growth	Implement streaming, add memory limits, use memory mapping
Network bandwidth	Monitor API request/response sizes and frequencies	Large prompts, inefficient batching, redundant API calls	Compress requests, optimize batch sizes, improve caching
Disk I/O	Profile file read/write patterns	Frequent checkpoint writes, large log files, cache thrashing	Use write-ahead logging, implement log rotation, optimize cache placement
API quota usage	Track requests per minute across all providers	Inefficient retry logic, redundant requests, poor load balancing	Implement intelligent retry policies, improve caching, add provider failover

Concurrency Optimization

The evaluation framework uses extensive concurrency to maximize throughput, but concurrency bugs can cause performance degradation or correctness issues that are difficult to diagnose.

Deadlock detection: Monitor for situations where evaluation progress stops completely despite no obvious errors. This often indicates deadlocks between components waiting for each other.

Resource contention analysis: Identify cases where concurrent operations compete for shared resources (cache locks, file handles, API rate limits) and cause performance degradation.

Batch size optimization: Different evaluation scenarios require different optimal batch sizes. Too small batches waste overhead on API calls and coordination; too large batches cause memory issues and reduce parallelism benefits.

Load balancing effectiveness: When using multiple LLM providers, monitor whether load is distributed evenly and whether failover logic works correctly under different failure scenarios.

Implementation Guidance

This section provides practical tools and code templates for implementing comprehensive debugging capabilities in the evaluation framework.

Technology Recommendations

Component	Simple Option	Advanced Option
Logging	Python <code>logging</code> module with structured JSON output	Distributed logging with ELK stack (Elasticsearch, Logstash, Kibana)
Metrics Collection	Simple counters and timers in application code	Prometheus metrics with Grafana dashboards
Error Tracking	File-based error logs with email alerts	Sentry for error aggregation and alerting
Performance Profiling	<code>cProfile</code> and <code>py-spy</code> for CPU profiling	APM tools like New Relic or DataDog
Interactive Debugging	Python debugger (<code>pdb</code>) with IPython integration	Remote debugging with PyCharm or VS Code
Log Analysis	grep/awk for simple log parsing	Structured log analysis with Pandas or specialized tools

Recommended File Structure

```
evaluation-framework/
├── src/evaluation_framework/
│   ├── debugging/
│   │   ├── __init__.py
│   │   ├── diagnostics.py      ← Main diagnostic utilities
│   │   ├── interactive_tools.py ← Interactive debugging interface
│   │   ├── log_analyzer.py    ← Log parsing and analysis
│   │   ├── performance_profiler.py ← Performance monitoring utilities
│   │   └── regression_detector.py ← Framework regression detection
│   ├── common/
│   │   ├── logging_config.py    ← Centralized logging configuration
│   │   └── error_types.py      ← Common error definitions
│   └── testing/
│       ├── regression_tests/   ← Automated regression test suite
│       ├── mock_providers.py    ← Mock LLM providers for testing
│       └── test_data_factory.py ← Utilities for generating test data
├── tools/
│   ├── debug_evaluation.py    ← Command-line debugging tool
│   ├── analyze_logs.py        ← Log analysis script
│   └── run_regression_tests.py ← Regression test runner
└── docs/
    └── debugging_runbook.md    ← Operational debugging procedures
```

Infrastructure Code: Logging and Diagnostics

Structured Logging Configuration

```
import logging

import json

import sys

from typing import Dict, Any, Optional

from datetime import datetime, timezone


class StructuredFormatter(logging.Formatter):

    """
    Custom formatter that outputs structured JSON logs for easier parsing and analysis.

    Includes correlation IDs, component names, and structured error information.
    """

    def format(self, record: logging.LogRecord) -> str:
        # TODO: Extract structured information from log record
        # TODO: Add correlation ID if present in record
        # TODO: Include component name and operation context
        # TODO: Format exception information as structured data
        # TODO: Serialize to JSON with consistent field names
        pass

    def setup_evaluation_logging(
        log_level: str = "INFO",
        log_file: Optional[str] = None,
        include_console: bool = True,
        correlation_id: Optional[str] = None
    ) -> logging.Logger:
        """
        Configure centralized logging for the evaluation framework.
        """
```

Args:

log_level: Minimum log level (DEBUG, INFO, WARNING, ERROR, CRITICAL)

```

    log_file: Optional file path for log output

    include_console: Whether to include console output

    correlation_id: Optional correlation ID for tracking requests

    Returns:
        Configured logger instance
    """
    # TODO: Create root logger for evaluation framework

    # TODO: Set up structured formatter with correlation ID

    # TODO: Add file handler if log_file specified

    # TODO: Add console handler if include_console is True

    # TODO: Configure log levels and formatting

    pass

class DiagnosticCollector:
    """
    Collects diagnostic information from all framework components.

    Provides a unified interface for health checking and troubleshooting.
    """

    def __init__(self):
        self._component_health_checkers: Dict[str, callable] = {}
        self._performance_monitors: Dict[str, callable] = {}
        self._error_collectors: Dict[str, callable] = {}

    def register_component_health_checker(self, component_name: str, health_checker: callable):
        """Register a health check function for a framework component."""
        # TODO: Validate health checker function signature
        # TODO: Store health checker with component name
        pass

```

```
def collect_system_diagnostics(self) -> Dict[str, Any]:  
    """  
    Collect comprehensive diagnostic information from all components.  
  
    Returns:  
        Dictionary containing health status, performance metrics, and error summaries  
    """  
  
    # TODO: Run all registered health checkers  
  
    # TODO: Collect performance metrics from monitors  
  
    # TODO: Aggregate error information from collectors  
  
    # TODO: Include system resource utilization  
  
    # TODO: Return structured diagnostic report  
  
    pass
```

Error Context and Recovery

```
from dataclasses import dataclass

from enum import Enum

from typing import List, Dict, Any, Optional, Callable

import traceback

import time

class ErrorSeverity(Enum):

    LOW = "low"

    MEDIUM = "medium"

    HIGH = "high"

    CRITICAL = "critical"

    @dataclass

    class ErrorContext:

        """Structured error information with recovery suggestions."""

        error_type: str

        severity: ErrorSeverity

        component: str

        operation: str

        evaluation_context: Dict[str, Any]

        recoverySuggestions: List[str]

        metadata: Dict[str, Any]

    class RecoveryStrategy:

        """

        Defines a strategy for recovering from specific error types.

        Each strategy encapsulates both detection logic and recovery actions.

        """

        def __init__(self, name: str, applicable_errors: List[str], recovery_action: Callable):

            self.name = name
```

```
    self.applicable_errors = applicable_errors

    self.recovery_action = recovery_action

    self.success_count = 0

    self.failure_count = 0


def is_applicable(self, error_context: ErrorContext) -> bool:
    """Check if this strategy applies to the given error context."""

    # TODO: Match error context against applicable_errors patterns

    # TODO: Consider error severity and component context

    # TODO: Return True if strategy should be attempted

    pass


def attempt_recovery(self, error_context: ErrorContext) -> bool:
    """
    Attempt to recover from the error using this strategy.

    Returns:
        True if recovery succeeded, False otherwise
    """

    # TODO: Execute recovery action with error context

    # TODO: Track success/failure statistics

    # TODO: Log recovery attempt and outcome

    # TODO: Return success status

    pass


class ErrorHandler:
    """
    Central error handling system that coordinates recovery strategies
    and provides debugging information for unrecoverable errors.
    """

```

```
def __init__(self):  
    self.recovery_strategies: List[RecoveryStrategy] = []  
    self.error_history: List[ErrorContext] = []  
    self.error_patterns: Dict[str, int] = {}  
  
def handle_error(self, exception: Exception, context: Dict[str, Any]) -> bool:  
    """  
    Handle an error with automatic recovery attempts.  
  
    Args:  
        exception: The exception that occurred  
        context: Additional context about the operation that failed  
  
    Returns:  
        True if error was recovered, False if manual intervention needed  
    """  
  
    # TODO: Create ErrorContext from exception and context  
    # TODO: Log structured error information  
    # TODO: Update error pattern statistics  
    # TODO: Try applicable recovery strategies in order  
    # TODO: Return recovery success status  
  
    pass
```

Core Logic Skeletons: Interactive Debugging Tools

Evaluation State Inspector

```
from typing import Dict, List, Any, Optional

import time

from dataclasses import asdict

class EvaluationStateInspector:

    """
    Interactive debugging interface for examining evaluation state.

    Provides real-time access to component status and performance metrics.
    """

    def __init__(self, coordinator: 'EvaluationCoordinator'):

        self.coordinator = coordinator

        self.last_snapshot_time: Optional[float] = None

        self.performance_history: List[Dict[str, Any]] = []

    def get_evaluation_overview(self) -> Dict[str, Any]:
        """
        Get high-level overview of all active evaluations.

        Returns:
            Dictionary with evaluation status, progress, and health information
        """

        # TODO: Query coordinator for active evaluation runs

        # TODO: Calculate overall progress and estimated completion times

        # TODO: Identify any stalled or failing evaluations

        # TODO: Include resource utilization summary

        # TODO: Return structured overview with actionable insights

        pass

    def inspect_evaluation_details(self, run_id: str) -> Dict[str, Any]:
```

```
"""
Get detailed information about a specific evaluation run.

Args:
    run_id: Unique identifier for the evaluation run

Returns:
    Detailed evaluation state including component status and performance

"""

# TODO: Retrieve evaluation run from coordinator

# TODO: Get current progress and batch processing status

# TODO: Analyze error patterns and failure rates

# TODO: Include cache hit rates and API performance

# TODO: Generate specific recommendations for optimization

pass


def diagnose_performance_issues(self, run_id: str) -> List[str]:
    """
Analyze evaluation performance and identify potential bottlenecks.

Args:
    run_id: Evaluation run to analyze

Returns:
    List of performance issues with suggested fixes

"""

# TODO: Analyze batch processing efficiency

# TODO: Check for API rate limiting or failures

# TODO: Examine cache hit rates and memory usage
```

```
# TODO: Identify concurrency issues or resource contention

# TODO: Generate prioritized list of optimization suggestions

pass

class MetricDebugger:

    """
    Interactive tool for testing and debugging individual metrics.

    Allows step-by-step execution with intermediate result inspection.
    """

    def __init__(self, metric_registry: 'MetricRegistry'):

        self.metric_registry = metric_registry

        self.debug_cache: Dict[str, Any] = {}

    def test_metric_isolated(
            self,
            metric_name: str,
            model_output: str,
            expected_output: str,
            test_case: 'TestCase'
        ) -> Dict[str, Any]:
        """
        Test a single metric in isolation with detailed debugging information.

        Args:
            metric_name: Name of metric to test
            model_output: Model response to evaluate
            expected_output: Expected/reference output
            test_case: Test case context
        """
        pass
```

```
    Returns:
        Detailed results including preprocessing, computation steps, and final score
    """
    # TODO: Retrieve metric implementation from registry
    # TODO: Trace preprocessing steps and intermediate transformations
    # TODO: Execute metric computation with timing information
    # TODO: Capture any errors or warnings during computation
    # TODO: Return comprehensive debugging information
    pass
```

```
def compare_metric_implementations(
    self,
    metric_names: List[str],
    model_output: str,
    expected_output: str,
    test_case: 'TestCase'
) -> Dict[str, Any]:
    """
    Compare multiple metrics on the same input to identify inconsistencies.
    """
    pass
```

```
    Returns:
        Comparison results with score differences and computation details
    """
    # TODO: Run all specified metrics on the same input
    # TODO: Compare preprocessing approaches and intermediate results
    # TODO: Analyze score distributions and identify outliers
    # TODO: Flag potential metric bugs or calibration issues
    # TODO: Generate recommendations for metric selection
    pass
```

Cache and Storage Debugging

```
import hashlib
import sqlite3
from pathlib import Path
from typing import Iterator, Tuple, Dict, Any, Optional

class CacheDebugger:
    """
    Utilities for diagnosing and repairing cache-related issues.

    Provides validation, integrity checking, and selective cache cleanup.
    """

    def __init__(self, cache_directory: Path):
        self.cache_directory = cache_directory
        self.validation_results: Dict[str, Any] = {}

    def validate_embedding_cache(self, provider: str, model: str) -> Dict[str, Any]:
        """
        Validate embedding cache integrity and consistency.

        Args:
            provider: Embedding provider name
            model: Model name to validate

        Returns:
            Validation results with any corruption or inconsistency reports
        """

        # TODO: Connect to embedding cache database
        # TODO: Validate that cached embeddings have correct dimensionality
        # TODO: Check for hash collisions in cache keys
        # TODO: Verify that embeddings are within expected value ranges

```

```
# TODO: Test a sample of embeddings by recomputing them
pass

def detect_cache_key_collisions(self) -> List[Tuple[str, List[str]]]:
    """
    Detect hash collisions in cache keys that could cause incorrect cache hits.

    Returns:
        List of (cache_key, original_texts) tuples showing collisions
    """
    # TODO: Scan all cache keys and group by hash value
    # TODO: For each hash, retrieve all original text inputs
    # TODO: Identify cases where different inputs map to same hash
    # TODO: Estimate probability and impact of collisions
    # TODO: Return collision report with examples
    pass
```

```
def repair_corrupted_cache_entries(self, corruption_report: Dict[str, Any]) -> bool:
    """
    Attempt to repair corrupted cache entries by recomputing values.

    Args:
        corruption_report: Results from cache validation
    Returns:
        True if repair succeeded, False if manual intervention needed
    """
    # TODO: Identify repairable vs non-repairable corruption
    # TODO: For repairable entries, recompute and update cache
```

Args:

corruption_report: Results from cache validation

Returns:

True if repair succeeded, False if manual intervention needed

"""

TODO: Identify repairable vs non-repairable corruption

TODO: For repairable entries, recompute and update cache

```
# TODO: Remove entries that cannot be repaired

# TODO: Update cache statistics and integrity metadata

# TODO: Verify repairs by re-running validation

pass

class DatasetVersionDebugger:

    """
    Debugging utilities for dataset versioning issues.

    Helps diagnose version conflicts, corruption, and merge issues.
    """

    def __init__(self, version_control: 'DatasetVersionControl'):

        self.version_control = version_control

    def validate_version_integrity(self, dataset_name: str) -> Dict[str, Any]:
        """
        Validate the integrity of a dataset's version history.

        Args:
            dataset_name: Name of dataset to validate

        Returns:
            Validation report with any detected issues
        """

        # TODO: Load all versions for the dataset

        # TODO: Verify parent-child relationships form valid DAG

        # TODO: Check content hash consistency for each version

        # TODO: Validate that diffs between versions are correct

        # TODO: Detect any orphaned or unreachable versions

        pass
```

```

def diagnose_merge_conflicts(
    self,
    dataset_name: str,
    version_a: str,
    version_b: str
) -> Dict[str, Any]:
    """
    Analyze merge conflicts between two dataset versions.

    Returns:
        Conflict analysis with suggested resolution strategies
    """
    # TODO: Load both versions and compute diffs
    # TODO: Identify conflicting test case modifications
    # TODO: Analyze conflict types (content vs metadata vs structure)
    # TODO: Generate merge suggestions based on conflict patterns
    # TODO: Provide interactive conflict resolution interface
    pass

```

Milestone Checkpoints

Milestone 1: Dataset Management Debugging

After implementing dataset management debugging tools, verify functionality with:

```

python -m evaluation_framework.debugging.diagnostics test-dataset-loading
python -m evaluation_framework.debugging.diagnostics validate-dataset-versions sample_dataset
python -m evaluation_framework.debugging.diagnostics check-schema-validation invalid_dataset.csv

```

Expected behaviors:

- Dataset loading errors provide specific column mapping suggestions
- Version validation detects and reports any integrity issues
- Schema validation failures include clear fix recommendations

- Performance profiling identifies bottlenecks in large dataset operations

Milestone 2: Metrics Engine Debugging

Test metric debugging capabilities:

```
python -m evaluation_framework.debugging.interactive_tools test-metric semantic_similarity "Hello world" "Hi there" BASH
python -m evaluation_framework.debugging.interactive_tools compare-metrics exact_match,fuzzy_match,semantic_similarity "Test output" "Expected output"
python -m evaluation_framework.debugging.diagnostics validate-embedding-cache openai text-embedding-3-small
```

Expected behaviors:

- Metric testing shows preprocessing steps and intermediate computations
- Metric comparisons highlight score differences and computation approaches
- Embedding cache validation detects corruption or inconsistencies
- LLM-as-judge debugging reveals prompt engineering issues

Milestone 3: Evaluation Runner Debugging

Validate evaluation execution debugging:

```
python -m evaluation_framework.debugging.diagnostics inspect-evaluation eval_run_12345 BASH
python -m evaluation_framework.debugging.performance_profiler analyze-concurrency eval_run_12345
python -m evaluation_framework.debugging.interactive_tools diagnose-performance-issues eval_run_12345
```

Expected behaviors:

- Evaluation inspection shows real-time progress and component status
- Concurrency analysis identifies rate limiting and resource contention
- Performance diagnosis provides specific optimization recommendations
- Cache debugging tools help resolve hit rate and consistency issues

Milestone 4: Reporting and Analysis Debugging

Test reporting system debugging:

```
python -m evaluation_framework.debugging.diagnostics validate-statistical-analysis eval_results.json BASH
python -m evaluation_framework.debugging.regression_detector check-framework-drift baseline_results.json current_results.json
python -m evaluation_framework.debugging.interactive_tools debug-report-generation eval_run_12345
```

Expected behaviors:

- Statistical analysis validation catches common errors (small samples, multiple comparisons)

- Framework drift detection distinguishes measurement issues from real performance changes
- Report generation debugging identifies visualization and export problems
- Regression detection helps maintain framework stability over time

Future Extensions

Milestone(s): All milestones (1-4) - Future extensions build upon the foundational capabilities established across dataset management (Milestone 1), metrics computation (Milestone 2), evaluation execution (Milestone 3), and reporting & analysis (Milestone 4)

Advanced Features

Think of the current evaluation framework as a solid foundation — like a well-built house with strong bones, plumbing, and electrical systems. The advanced features are sophisticated additions that transform this functional house into a smart home with automation, predictive maintenance, and seamless integration with the broader neighborhood infrastructure. Each advanced feature leverages the core capabilities we've built while extending them in ways that weren't feasible to include in the initial design due to complexity constraints.

The advanced features fall into three categories: **enhanced evaluation capabilities** that make assessments more sophisticated and nuanced, **intelligent automation** that reduces manual oversight and intervention, and **scale optimization** that enables evaluation of massive datasets and complex model architectures. Each category builds incrementally on the existing system without requiring fundamental architectural changes.

Multi-Modal Evaluation Support

The current framework focuses exclusively on text-to-text evaluation, but modern LLM applications increasingly involve multi-modal inputs and outputs. Think of this extension as adding support for evaluating a conversation that includes not just words, but also images, audio, and structured data — like assessing how well an AI assistant can analyze a medical image, interpret a patient's spoken symptoms, and generate both a written diagnosis and a visual explanation.

Decision: Multi-Modal Architecture

- **Context:** Modern LLMs handle text, images, audio, and structured data, but current evaluation framework only supports text inputs and outputs
- **Options Considered:**
 1. Separate evaluation frameworks for each modality
 2. Universal evaluation interface with modality-specific implementations
 3. Pipeline-based evaluation with modality conversion stages
- **Decision:** Universal evaluation interface with modality-specific metric implementations
- **Rationale:** Maintains consistent evaluation semantics while allowing specialized handling of different data types; enables cross-modal evaluation scenarios
- **Consequences:** Requires extending `TestCase` schema and developing modality-specific metrics, but preserves existing evaluation workflow

The multi-modal extension introduces several new components that integrate seamlessly with the existing architecture. The `MultiModalTestCase` extends the current `TestCase` structure to handle different input and output modalities while maintaining backward compatibility with text-only evaluations.

Component	Purpose	Integration Point
<code>MultiModalTestCase</code>	Test case supporting images, audio, video inputs	Extends existing <code>TestCase</code> with modality fields
<code>ModalityProcessor</code>	Converts between different data formats	Plugs into preprocessing pipeline
<code>CrossModalMetric</code>	Measures alignment between different modalities	Implements <code>BaseMetric</code> interface
<code>ModalityCache</code>	Caches expensive modality conversions	Extends existing caching infrastructure

The evaluation workflow adapts naturally to handle multi-modal data. When a test case contains an image input, the `ModalityProcessor` converts it to embeddings or other representations that metrics can consume. For example, evaluating an image captioning model involves comparing generated text against reference captions using semantic similarity, while also checking whether the caption accurately describes visual elements present in the image.

Evaluation Type	Input Modality	Output Modality	Metric Examples
Image Captioning	Image	Text	CLIP similarity, object detection accuracy
Audio Transcription	Audio	Text	Word error rate, speaker identification
Visual Question Answering	Image + Text	Text	Answer accuracy, visual grounding
Document Analysis	PDF/Image	Structured JSON	Field extraction accuracy, layout preservation

Adaptive Evaluation Strategies

Current evaluation runs all test cases through the same metrics regardless of case difficulty or characteristics. An adaptive evaluation system acts like an experienced teacher who adjusts questioning strategy based on student performance — easy questions get quick verification, while challenging cases receive deeper analysis with multiple evaluation approaches.

The adaptive strategy engine analyzes test case metadata, model confidence scores, and historical performance patterns to dynamically select appropriate metrics and evaluation depth. This significantly reduces evaluation cost while maintaining accuracy for cases that require detailed assessment.

Decision: Adaptive Metric Selection

- **Context:** Running expensive LLM-as-judge evaluation on every test case is costly, but some cases need minimal verification while others require deep analysis
- **Options Considered:**
 1. Static metric assignment based on test case tags
 2. Model confidence-based metric selection
 3. Multi-stage evaluation with escalation rules
- **Decision:** Multi-stage evaluation with confidence-based escalation
- **Rationale:** Balances cost and accuracy by using cheap metrics first, escalating to expensive evaluation only when needed
- **Consequences:** Requires confidence scoring infrastructure and escalation rule engine, but can reduce evaluation costs by 60-80%

The adaptive evaluation pipeline operates in three stages. **Stage 1** applies fast, deterministic metrics like exact match and BLEU scores to all test cases. **Stage 2** escalates cases with low confidence or conflicting signals to semantic similarity evaluation. **Stage 3** applies expensive LLM-as-judge evaluation only to cases where automated metrics provide insufficient signal.

Stage	Criteria	Metrics Applied	Cost Factor
Stage 1	All cases	Exact match, BLEU, ROUGE	1x
Stage 2	Confidence < 0.8 OR conflicting signals	Semantic similarity, embedding-based	10x
Stage 3	High-stakes cases OR ambiguous results	LLM-as-judge, human-in-loop	100x

The escalation rules are configurable and can incorporate domain-specific knowledge. For instance, mathematical reasoning problems might escalate based on numerical accuracy, while creative writing tasks might escalate based on semantic diversity from the reference answer.

Continuous Learning from Evaluation Results

Think of this feature as giving the evaluation system a memory and learning capability — like a quality assurance inspector who gets better at spotting defects by analyzing patterns in past inspections. The continuous learning system analyzes evaluation results over time to improve metric calibration, detect evaluation blind spots, and recommend dataset improvements.

The learning system operates on three timescales. **Real-time learning** adjusts metric weights and thresholds during evaluation runs based on observed score distributions. **Session learning** analyzes completed evaluation runs to identify systematic biases or metric inconsistencies. **Historical learning** examines long-term trends to suggest dataset evolution and metric retirement strategies.

Learning Type	Timescale	Adaptation Target	Example Improvement
Real-time	During evaluation	Metric thresholds, batch sizing	Adjust semantic similarity threshold based on score distribution
Session	Post-evaluation	Metric correlations, failure patterns	Identify metrics that consistently disagree and investigate
Historical	Weekly/monthly	Dataset composition, metric retirement	Recommend new test cases for under-represented failure modes

The learning system maintains a `MetricPerformanceHistory` that tracks metric reliability, correlation patterns, and prediction accuracy over time. This data feeds into automatic recalibration algorithms and human-readable improvement recommendations.

Hierarchical Evaluation Taxonomies

Current evaluation treats all test cases uniformly, but real-world applications have hierarchical skill requirements. Think of this like academic assessment — you need to master basic arithmetic before attempting calculus, and evaluation should reflect these skill dependencies. The hierarchical evaluation system organizes test cases into skill trees and provides detailed breakdowns of model capabilities across different competency levels.

The taxonomy system introduces `SkillNode` entities that represent different capabilities and their dependencies. A language model's performance on "mathematical reasoning" might depend on its mastery of "numerical calculation," "logical inference," and "word problem parsing." The evaluation system tracks performance at each skill level and identifies capability gaps.

Skill Level	Dependencies	Evaluation Focus	Progression Criteria
Basic Skills	None	Fundamental capabilities	95%+ accuracy on core tasks
Intermediate	2-3 basic skills	Skill combination	85%+ accuracy with reasoning trace
Advanced	Multiple intermediate	Complex problem solving	70%+ accuracy on novel problems
Expert	All prerequisite skills	Domain specialization	Human-level performance on expert tasks

The hierarchical analysis generates **skill gap reports** that identify which foundational capabilities need improvement before attempting more complex evaluations. This is particularly valuable for model training guidance and capability assessment for production deployment.

Distributed Evaluation Infrastructure

As datasets grow to millions of test cases and models become more expensive to query, evaluation needs to scale horizontally across multiple machines and regions. Think of this as transforming evaluation from a single powerful workstation to a distributed computing cluster — like moving from a master craftsman's workshop to a modern factory with specialized stations and assembly lines.

The distributed evaluation system introduces a **coordinator-worker architecture** where a central coordinator partitions datasets and distributes work to evaluation workers running on different machines. Workers can be heterogeneous —

some optimized for fast exact match computation, others equipped with GPUs for semantic similarity, and specialized workers for expensive LLM-as-judge evaluation.

Component	Responsibility	Scaling Characteristics
EvaluationCoordinator	Work distribution, result aggregation	Single instance with failover
EvaluationWorker	Metric computation, LLM queries	Horizontally scalable
ResultAggregator	Statistical analysis, report generation	CPU-intensive, vertically scalable
CacheCluster	Distributed result caching	Horizontally scalable with sharding

The distributed system maintains **evaluation consistency** through deterministic work partitioning and result checksums. If a worker fails, its assigned test cases are redistributed to healthy workers with automatic deduplication to prevent double-evaluation.

Decision: Work Distribution Strategy

- **Context:** Large evaluations need to distribute work across multiple machines while maintaining result consistency and handling worker failures
- **Options Considered:**
 1. Static work assignment based on test case hash
 2. Dynamic load balancing with work stealing
 3. Hierarchical distribution with regional coordinators
- **Decision:** Dynamic load balancing with consistent hashing for cache locality
- **Rationale:** Adapts to heterogeneous worker capabilities and provides natural cache affinity while handling failures gracefully
- **Consequences:** Requires sophisticated coordinator logic but provides optimal resource utilization and fault tolerance

Real-Time Evaluation Monitoring

Production LLM applications need continuous evaluation as they serve live traffic, similar to how modern web applications use real-user monitoring (RUM) to track performance. The real-time evaluation system samples live model responses, applies lightweight evaluation metrics, and alerts when quality degrades below acceptable thresholds.

The monitoring system operates with **configurable sampling rates** to balance evaluation coverage with computational overhead. High-confidence responses get minimal evaluation, while responses showing uncertainty indicators receive more thorough assessment. This creates a continuous feedback loop that can detect model degradation, prompt injection attacks, or training distribution drift.

Monitoring Type	Sampling Rate	Latency Tolerance	Alert Criteria
Quality Drift	1-5% of traffic	< 100ms additional latency	10% degradation over 1 hour
Safety Violations	100% of flagged responses	< 50ms additional latency	Any detected violation
Performance Regression	0.1% of traffic	< 1s additional latency	Statistical significance test

The real-time system integrates with the existing evaluation infrastructure by treating live traffic as a continuously updating dataset. Evaluation results feed into the same reporting and analysis pipeline, enabling unified views of both batch evaluation and production performance.

Integration Points

The evaluation framework's true power emerges when it integrates seamlessly with the broader machine learning infrastructure ecosystem. Think of these integration points as diplomatic relationships between neighboring countries — each system maintains its sovereignty and core competencies while establishing protocols for collaboration, data exchange, and mutual support.

MLOps Pipeline Integration

Modern machine learning development follows DevOps principles with continuous integration, automated testing, and staged deployments. The evaluation framework integrates into this pipeline as a **quality gate** — automated evaluation runs must pass before model deployments can proceed to production. This prevents degraded models from reaching users while maintaining development velocity.

The MLOps integration operates through standardized interfaces and event-driven communication. When a new model version is committed to the model registry, it triggers an automated evaluation run against the comprehensive test suite. The evaluation results feed into deployment decision logic that can automatically promote models that exceed quality thresholds or flag models requiring human review.

Pipeline Stage	Evaluation Trigger	Pass Criteria	Failure Action
Development	Code commit with model changes	Basic functionality tests pass	Block merge request
Staging	Model registry update	Comprehensive evaluation > baseline	Hold deployment, notify team
Production	Scheduled evaluation	No regression detected	Automatic rollback to previous version
Post-deployment	Live traffic monitoring	Quality metrics within bounds	Alert and gradual traffic reduction

The integration provides **evaluation-aware deployment strategies** that can gradually shift traffic to new model versions while continuously monitoring quality metrics. If evaluation detects degradation, the system can automatically roll back or reduce traffic to the problematic model version.

Decision: Deployment Integration Strategy

- **Context:** Model deployments need automated quality gates without blocking development velocity or requiring manual intervention for routine deployments
- **Options Considered:**
 1. Blocking evaluation that prevents deployment until completion
 2. Asynchronous evaluation with post-deployment rollback
 3. Staged deployment with continuous evaluation
- **Decision:** Staged deployment with continuous evaluation and automatic traffic management
- **Rationale:** Balances safety with velocity by detecting issues in production while limiting impact through controlled traffic exposure
- **Consequences:** Requires sophisticated traffic management and rollback capabilities but provides maximum protection with minimal development friction

Model Registry Integration

The model registry serves as the central repository for trained models, their metadata, and versioning information. The evaluation framework integrates deeply with the registry to provide **evaluation-aware model management** — every model version includes comprehensive evaluation results, and models can be queried based on performance characteristics rather than just chronological order.

The integration automatically tags model versions with evaluation metadata including overall scores, per-category breakdowns, and regression analysis results. This enables sophisticated model selection strategies that consider not just overall performance but specific capability requirements for different use cases.

Model Metadata	Source	Usage
Overall evaluation score	Comprehensive evaluation run	Model ranking and selection
Per-category performance	Tag-based evaluation breakdown	Domain-specific model selection
Regression analysis	Comparison with baseline models	Deployment risk assessment
Evaluation confidence	Statistical analysis of results	Decision threshold adjustment
Resource requirements	Evaluation runtime profiling	Cost-aware model selection

The registry integration enables **capability-based model queries** where applications can request models meeting specific performance criteria. For example, a customer service application might require models with >0.9 accuracy on "customer_intent_classification" and >0.8 on "response_empathy" metrics.

Experimentation Platform Integration

A/B testing platforms manage controlled experiments that compare different model versions or prompt strategies with live user traffic. The evaluation framework integrates with these platforms to provide **offline evaluation correlation** with online performance metrics. This helps validate that offline evaluation accurately predicts real-world performance and identifies when offline metrics become unreliable.

The integration operates through shared experiment identifiers that link offline evaluation runs with online experiment results. Statistical analysis compares offline evaluation predictions with observed user satisfaction, task completion rates, and business metrics to continuously calibrate evaluation accuracy.

Integration Point	Data Flow	Purpose
Experiment Design	Evaluation results → Experiment variants	Use evaluation to pre-filter experimental candidates
Online Monitoring	Live metrics → Evaluation correlation analysis	Validate offline evaluation accuracy
Result Analysis	Combined offline/online results	Comprehensive performance assessment
Model Selection	Correlated metrics → Production deployment	Choose models optimized for real-world performance

The platform provides **evaluation-guided experimentation** that uses offline evaluation results to inform experiment design. Instead of randomly testing model variants, the system can design experiments that focus on cases where offline evaluation showed uncertainty or disagreement between metrics.

Data Pipeline Integration

Modern ML systems process continuous streams of training data, user interactions, and feedback signals. The evaluation framework integrates with data pipelines to enable **continuous dataset evolution** — new test cases are automatically generated from production data, golden examples are identified from high-confidence interactions, and dataset composition adapts to emerging use patterns.

The data pipeline integration monitors live system interactions to identify evaluation gaps. When users consistently encounter scenarios not covered in the existing test suite, the system flags these for potential inclusion in future dataset versions. This creates a feedback loop that keeps evaluation datasets current with real-world usage patterns.

Data Source	Processing	Output
User interactions	Pattern analysis, quality filtering	Candidate test cases
Human feedback	Confidence scoring, consensus detection	Golden examples
Error logs	Failure pattern analysis	Stress test scenarios
Performance metrics	Correlation analysis	Evaluation validation data

The integration provides **automated dataset curation** that reduces the manual effort required to maintain comprehensive evaluation coverage. The system can automatically propose new test cases, identify outdated examples, and suggest dataset rebalancing based on observed performance patterns.

Monitoring and Alerting Integration

Production monitoring systems track system health, performance metrics, and business KPIs. The evaluation framework integrates with these systems to provide **quality-aware alerting** that can distinguish between system performance issues and model quality degradation. This prevents false alerts when response times are normal but output quality has declined.

The monitoring integration establishes evaluation metrics as first-class observability signals alongside traditional infrastructure metrics. Quality degradation alerts include context about which specific capabilities have declined and recommended remediation actions based on historical patterns.

Alert Type	Trigger Condition	Context Provided	Recommended Action
Quality Regression	Statistical significance test failure	Affected categories, magnitude	Model rollback or retraining
Safety Violation	Any harmful content detection	Specific safety category	Immediate human review
Performance Drift	Gradual decline over time window	Trend analysis, root cause hints	Dataset refresh or fine-tuning
Evaluation System Health	Evaluation pipeline failures	Component status, error patterns	Infrastructure investigation

The integration enables **proactive quality management** where teams receive early warning of potential issues before they impact user experience significantly. Combined with automated remediation capabilities, this creates a self-healing evaluation and deployment system.

Implementation Guidance

Building these advanced features requires careful architectural planning and incremental development. The key principle is extending the existing system's interfaces rather than replacing core components, which maintains backward compatibility while adding sophisticated new capabilities.

Technology Recommendations

Feature Category	Simple Approach	Advanced Approach
Multi-Modal Support	File-based modality storage with conversion utilities	Distributed object storage with streaming processors
Adaptive Evaluation	Rule-based metric selection with configuration files	Machine learning-based optimization with reinforcement learning
Continuous Learning	Periodic batch analysis with manual threshold updates	Real-time streaming analysis with automatic adaptation
Distributed Infrastructure	Docker Compose multi-container setup	Kubernetes with custom operators and auto-scaling
MLOps Integration	Webhook-based triggers with REST API integration	Event-driven architecture with message queues

Recommended Project Structure

```
evaluation-framework/
  extensions/
    multi_modal/
      processors/
        image_processor.py
        audio_processor.py
        document_processor.py
    metrics/
      cross_modal_similarity.py
      visual_grounding.py
    test_cases/
      multi_modal_test_case.py
adaptive/
  strategy_engine.py
  confidence_scorer.py
  escalation_rules.py
learning/
  metric_calibration.py
  performance_tracking.py
  improvement_recommendations.py
distributed/
  coordinator.py
  worker.py
  result_aggregator.py
integrations/
  mlops_connector.py
  model_registry_client.py
  monitoring_adapter.py
```

Multi-Modal Infrastructure Starter Code

```
from abc import ABC, abstractmethod

from typing import Dict, Any, List, Union

import numpy as np

from pathlib import Path

class ModalityProcessor(ABC):

    """Base class for processing different data modalities."""

    @abstractmethod
    def process(self, input_data: Any) -> Dict[str, Any]:
        """Process input data and return structured representation."""
        pass

    @abstractmethod
    def supported_formats(self) -> List[str]:
        """Return list of supported file formats."""
        pass

class ImageProcessor(ModalityProcessor):

    """Processor for image inputs using computer vision models."""

    def __init__(self, model_name: str = "clip-vit-base-patch32"):
        self.model_name = model_name
        # TODO: Initialize vision model for feature extraction
        # TODO: Setup image preprocessing pipeline

    def process(self, image_path: Path) -> Dict[str, Any]:
        """Extract features and metadata from image."""
        # TODO: Load and preprocess image
        # TODO: Extract visual features using vision model
```

PYTHON

```
# TODO: Detect objects and generate descriptions

# TODO: Return structured representation with embeddings

pass


def supported_formats(self) -> List[str]:
    return ['.jpg', '.jpeg', '.png', '.bmp', '.tiff']


class AudioProcessor(ModalityProcessor):

    """Processor for audio inputs using speech recognition."""

    def __init__(self, model_name: str = "whisper-base"):

        self.model_name = model_name

        # TODO: Initialize speech recognition model

        # TODO: Setup audio preprocessing pipeline


    def process(self, audio_path: Path) -> Dict[str, Any]:
        """Transcribe audio and extract acoustic features."""

        # TODO: Load and preprocess audio file

        # TODO: Transcribe speech to text

        # TODO: Extract acoustic features (pitch, tempo, etc.)

        # TODO: Return structured representation with transcription

        pass


    def supported_formats(self) -> List[str]:
        return ['.wav', '.mp3', '.m4a', '.flac']
```

Adaptive Evaluation Core Logic

```
from typing import List, Dict, Tuple, Optional
from enum import Enum

class EvaluationStage(Enum):
    FAST = "fast_metrics"
    SEMANTIC = "semantic_analysis"
    COMPREHENSIVE = "llm_judge"

class AdaptiveEvaluationEngine:
    """Engine that selects appropriate evaluation depth based on test case characteristics."""

    def __init__(self, config: Dict[str, Any]):
        self.confidence_threshold = config.get('confidence_threshold', 0.8)
        self.cost_budget = config.get('cost_budget', 1000.0)

        # TODO: Initialize metric registry with cost annotations
        # TODO: Setup escalation rules configuration
        # TODO: Initialize confidence scoring model

    def evaluate_test_case(self, test_case: TestCase, model_response: str) -> MetricResult:
        """Apply adaptive evaluation strategy to single test case."""

        results = []

        # TODO 1: Apply fast metrics (exact match, BLEU) to get initial signal
        # TODO 2: Calculate confidence score based on fast metric agreement
        # TODO 3: Check escalation criteria (confidence, test case difficulty, cost budget)
        # TODO 4: If escalation needed, apply semantic similarity metrics
        # TODO 5: If still uncertain, apply LLM-as-judge evaluation
        # TODO 6: Combine results from all stages into final MetricResult
        # TODO 7: Update cost tracking and confidence calibration
```

```
pass

def should_escalate(self, stage: EvaluationStage, results: Dict[str, float],
                     test_case: TestCase) -> bool:
    """Determine if evaluation should proceed to next stage."""

    # TODO 1: Check confidence threshold - if results agree, stop escalation

    # TODO 2: Check cost budget - if exceeded, stop escalation

    # TODO 3: Check test case priority - high stakes cases always escalate

    # TODO 4: Check metric disagreement - conflicting signals trigger escalation

    # TODO 5: Return boolean decision with logging for audit trail

pass
```

Distributed Evaluation Coordinator

```
import asyncio  
  
from typing import Dict, List, Set  
  
import uuid  
  
  
class EvaluationCoordinator:  
  
    """Coordinates distributed evaluation across multiple worker nodes."""  
  
  
    def __init__(self, worker_endpoints: List[str]):  
  
        self.worker_endpoints = worker_endpoints  
  
        self.active_evaluations: Dict[str, EvaluationRun] = {}  
  
        self.work_queue = asyncio.Queue()  
  
        # TODO: Initialize worker health monitoring  
  
        # TODO: Setup result aggregation pipeline  
  
        # TODO: Configure failure detection and recovery  
  
  
    async def run_distributed_evaluation(self, dataset: Dataset,  
  
                                         metrics: List[str]) -> EvaluationRun:  
  
        """Execute evaluation across distributed worker pool."""  
  
        evaluation_id = str(uuid.uuid4())  
  
  
        # TODO 1: Partition dataset into work units based on worker capabilities  
  
        # TODO 2: Create evaluation run record with distributed metadata  
  
        # TODO 3: Distribute work units to available workers with load balancing  
  
        # TODO 4: Monitor progress and handle worker failures with redistribution  
  
        # TODO 5: Collect and aggregate results as they complete  
  
        # TODO 6: Generate final evaluation report with timing and resource usage  
  
        # TODO 7: Cleanup temporary resources and update evaluation registry  
  
  
        pass
```

```
async def monitor_worker_health(self):

    """Continuously monitor worker availability and performance."""

    # TODO 1: Send periodic health check requests to all workers

    # TODO 2: Track response times and error rates per worker

    # TODO 3: Remove unhealthy workers from active pool

    # TODO 4: Redistribute work from failed workers to healthy ones

    # TODO 5: Log worker performance metrics for capacity planning

    pass
```

Milestone Checkpoints

Multi-Modal Extension Checkpoint: After implementing multi-modal support, verify:

- `python -m pytest tests/test_multi_modal.py` passes all tests
- Load image test case: `evaluation_framework multi-modal --input sample.jpg --expected "A cat sitting on a chair"`
- Expected output: Evaluation results with visual grounding score and object detection metrics

Adaptive Evaluation Checkpoint: After implementing adaptive strategies, verify:

- Run evaluation with cost limits: `evaluation_framework evaluate --adaptive --cost-budget 100`
- Check that expensive metrics are only applied to uncertain cases
- Expected output: <30% of test cases use LLM-as-judge evaluation while maintaining accuracy

Distributed Evaluation Checkpoint: After implementing distributed infrastructure, verify:

- Start coordinator: `evaluation_framework coordinator --workers 3`
- Run distributed evaluation: `evaluation_framework evaluate --distributed --dataset large_test.json`
- Expected output: Work distributed across workers with automatic failure recovery

Debugging Advanced Features

Symptom	Likely Cause	Diagnosis	Fix
Multi-modal metrics return NaN	Unsupported file format or corrupted media	Check file format validation logs	Add format conversion or fix input pipeline
Adaptive evaluation always escalates	Confidence thresholds too high	Review confidence score distribution	Calibrate thresholds based on metric agreement patterns
Distributed evaluation hangs	Worker health check failures	Check network connectivity and worker logs	Implement exponential backoff and worker replacement
Integration tests fail after extension	Breaking changes to core interfaces	Run interface compatibility checks	Use adapter pattern to maintain backward compatibility

The advanced features transform the evaluation framework from a functional tool into a comprehensive evaluation platform. Each extension builds incrementally on the core architecture while opening new possibilities for sophisticated LLM assessment and integration with modern ML infrastructure.

Glossary

Milestone(s): All milestones (1-4) - Glossary provides unified terminology and definitions used throughout dataset management (Milestone 1), metrics computation (Milestone 2), evaluation execution (Milestone 3), and reporting & analysis (Milestone 4)

Think of this glossary as the universal translator for LLM evaluation. Just as medical professionals need precise definitions for terms like "tachycardia" or "myocardial infarction" to communicate effectively, LLM evaluation practitioners need shared understanding of concepts like "semantic similarity" or "LLM-as-judge." Without consistent terminology, a team member's "fuzzy matching" might mean something completely different from another's, leading to implementation confusion and architectural misalignment.

Core Evaluation Concepts

Evaluation Excellence: The overarching philosophy of delivering superior evaluation capabilities that provide reliable, actionable insights into LLM performance. This goes beyond simple accuracy measurement to encompass comprehensive understanding of model strengths, weaknesses, and behavioral patterns. Evaluation excellence requires statistical rigor, reproducible methodologies, and clear communication of results to both technical and business stakeholders.

Test Case Schema: The structured format that defines evaluation test cases with comprehensive metadata. Each test case follows a consistent schema that includes the input prompt, expected output, categorization tags, difficulty level, and extensible metadata fields. The schema ensures that evaluation datasets remain interoperable across different evaluation runs and can be processed consistently by all metrics.

Field	Type	Description
case_id	str	Unique identifier for the test case within the dataset
prompt	str	Input text or query that will be sent to the LLM
expected_output	str	Reference answer or expected response for comparison
tags	List[str]	Categorical labels for grouping and filtering test cases
difficulty	str	Complexity level indicator (easy, medium, hard, expert)
metadata	Dict	Extensible key-value pairs for additional test case information

Golden Examples: High-quality reference cases specifically curated for calibrating evaluators, particularly LLM-as-judge systems. Golden examples represent the gold standard for expected model behavior and are typically created through expert review or consensus among multiple human evaluators. These examples serve as anchor points for establishing consistent evaluation criteria and are often used to validate that automated metrics are producing reasonable scores.

Golden examples are like the master craftsman's reference pieces in a workshop - they define what excellence looks like and help calibrate all other evaluation tools to maintain consistent standards.

Evaluation Results Model: The comprehensive data structure that captures complete evaluation outcomes, including not just scores but also metadata about the evaluation process itself. This model preserves full traceability from input test case through metric computation to final aggregated results.

Field	Type	Description
case_id	str	Reference to the original test case identifier
model_response	str	Actual output generated by the LLM under evaluation
response_metadata	Dict	Information about response generation (tokens, latency, etc.)
metric_results	Dict	Per-metric scores and detailed evaluation information
overall_score	float	Weighted combination of all applicable metric scores
evaluation_time	datetime	Timestamp when evaluation was completed
error_info	Optional[str]	Error details if evaluation failed

Dataset Management Terminology

Dataset Versioning: A git-like tracking system for changes to evaluation datasets that maintains complete history and enables rollback to previous states. This system tracks not only which test cases were added, removed, or modified, but also preserves the full context of why changes were made through commit messages and change summaries.

Version Diff: A detailed comparison showing the specific changes between two dataset versions, including added cases, modified cases, removed cases, and field-level changes within individual test cases. Version diffs enable reviewers to understand the impact of dataset updates and ensure that changes align with evaluation objectives.

Field	Type	Description
from_version	str	Source version identifier in the comparison
to_version	str	Target version identifier in the comparison
added_cases	List[str]	Test case IDs that exist only in the target version
modified_cases	List[str]	Test case IDs with content changes between versions
removed_cases	List[str]	Test case IDs that exist only in the source version
case_changes	Dict[str, Dict[str, Any]]	Field-level changes for each modified test case

Content Hash: A cryptographic hash function output that serves as a unique identifier based on dataset content rather than arbitrary version numbers. Content hashing ensures data integrity by detecting any unauthorized modifications and enables content-addressable storage where identical datasets share the same hash regardless of their creation time or source.

Merge Conflict: A situation that occurs when the same test case has been modified differently in parallel branches of dataset development, requiring manual resolution to determine the canonical version. The system detects conflicts by comparing field-level changes and provides tools for reviewers to make informed decisions about conflict resolution.

Stratified Sampling: A statistical sampling technique that preserves the proportional representation of different subgroups when splitting datasets into training, validation, and test sets. For evaluation datasets, stratification typically occurs across difficulty levels, topic categories, or other meaningful dimensions to ensure that each subset remains representative of the overall population.

Metrics and Scoring Terminology

Semantic Similarity: A measurement technique that compares the meaning similarity between model output and reference text using embedding vectors rather than surface-level text comparison. This approach captures conceptual similarity even when the exact wording differs, making it particularly valuable for evaluating tasks where multiple valid phrasings exist.

LLM-as-Judge: An evaluation methodology that uses language models to assess the quality of other language model outputs, typically by providing structured prompts that ask the judge model to score responses according to specific criteria. This approach enables evaluation of complex, subjective qualities that are difficult to capture with traditional metrics.

Fuzzy Matching: Approximate string matching that applies various normalization techniques (whitespace removal, case normalization, punctuation handling) before comparison to reduce the impact of trivial formatting differences on evaluation scores. Fuzzy matching bridges the gap between overly strict exact matching and computationally expensive semantic similarity.

Metric Applicability: Rules and logic that determine which metrics should be applied to evaluate specific test cases, based on test case metadata, tags, or content characteristics. Not all metrics are meaningful for all types of evaluation tasks, so applicability rules prevent inappropriate metric application and ensure evaluation results remain interpretable.

Score Normalization: The process of mapping diverse metric outputs to a standardized 0.0-1.0 range to enable meaningful aggregation and comparison across different metric types. Normalization preserves the relative ranking of results while ensuring that metrics with different natural scales can be combined effectively.

Judge Prompt Template: A structured format for creating consistent prompts when using LLM-as-judge evaluation, including standardized sections for task description, evaluation criteria, output format specifications, and example demonstrations. Templates ensure that judge evaluations remain consistent across different test cases and evaluation runs.

Consistency Control: Mechanisms implemented to improve the reliability and repeatability of LLM-as-judge evaluations, such as temperature control, multiple sampling with majority voting, or chain-of-thought reasoning requirements. These controls help mitigate the inherent variability in language model outputs.

Calibration: The process of mapping raw similarity scores or other metric outputs to meaningful evaluation scores that align with human judgment or established quality standards. Calibration involves analyzing the distribution of raw scores against known-good examples and establishing thresholds that correspond to different quality levels.

Execution and Processing Terminology

Batch Processing: A processing strategy that groups test cases into batches for parallel execution, optimizing throughput while respecting resource constraints and API rate limits. Batch processing enables efficient utilization of computational resources and helps manage the trade-off between evaluation speed and system stability.

Result Caching: A storage system that persists LLM responses and computed metric results to avoid redundant computation when the same inputs are evaluated multiple times. Caching is particularly valuable for expensive operations like LLM API calls and embedding computations.

Content-Addressable Cache: A caching system where cache keys are derived from the content being cached rather than arbitrary identifiers, ensuring that identical content always maps to the same cache entry regardless of when or how it was generated. This approach maximizes cache hit rates and eliminates duplicate storage.

Checkpointing: The practice of periodically saving evaluation progress to persistent storage, enabling recovery and resumption after system failures or interruptions. Checkpoints include both the current state of the evaluation and enough metadata to resume processing from the exact point of interruption.

Write-Ahead Log (WAL): A transaction logging mechanism that records intended operations before executing them, providing a reliable foundation for crash recovery. The WAL ensures that no evaluation progress is lost even if the system crashes in the middle of processing a batch of test cases.

Progress Tracking: Real-time monitoring and reporting of evaluation completion status, including completed case counts, estimated time remaining, error rates, and performance metrics. Progress tracking provides visibility into long-running evaluations and helps identify performance bottlenecks.

Adaptive Batch Sizing: A dynamic optimization technique that adjusts batch sizes based on observed performance characteristics, increasing batch sizes when the system is performing well and reducing them when errors or slowdowns are detected. This approach maximizes throughput while maintaining system stability.

Rate Limiting: Control mechanisms that regulate the frequency of API requests to prevent overwhelming external services and avoid triggering rate limit responses. Rate limiting includes both simple request counting and more sophisticated algorithms like token bucket or sliding window approaches.

Analysis and Reporting Terminology

Score Aggregation: The process of computing statistical summaries across evaluation results, including measures of central tendency (mean, median), variability (standard deviation, percentiles), and group-based breakdowns by tags.

categories, or other metadata dimensions.

Statistical Significance: A statistical concept that quantifies the likelihood that an observed difference between two sets of evaluation results is due to genuine performance differences rather than random variation. Statistical significance testing helps distinguish meaningful changes from noise in evaluation comparisons.

Effect Size: A measure of the practical magnitude of performance differences that is independent of sample size, providing insight into whether statistically significant differences are also meaningfully large. Effect size helps prioritize which performance changes deserve attention and resources.

Regression Detection: Automated identification of performance degradation compared to established baseline measurements, using statistical techniques to distinguish genuine regressions from normal variation. Regression detection enables proactive identification of quality issues before they impact production systems.

Changepoint Detection: Statistical analysis that identifies moments in time when the fundamental performance characteristics of a model shift, indicating potential changes in model behavior, data distribution, or evaluation methodology. Changepoint detection helps understand the timeline of model performance evolution.

Failure Analysis: Systematic examination of evaluation failures to identify common patterns, root causes, and remediation opportunities. Failure analysis involves clustering similar failures, analyzing error patterns, and generating actionable recommendations for model improvement.

Semantic Clustering: The process of grouping evaluation failures or results based on meaning similarity using embedding-based approaches, enabling identification of thematic patterns that might not be apparent through simple keyword-based analysis.

Root Cause Analysis: A systematic process for tracing failure patterns back to their underlying model limitations, training data issues, or architectural problems, providing actionable insights for model improvement rather than just symptom identification.

Baseline Management: The practice of maintaining reference performance levels for comparison purposes, including policies for when baselines should be updated and how to handle the evolution of evaluation standards over time.

Multiple Comparison Correction: Statistical adjustments applied when testing multiple hypotheses simultaneously to control the overall probability of false discoveries. This is particularly important when comparing performance across many different categories or metrics within a single evaluation.

System Architecture and Infrastructure

Plugin System: An extensible architecture that allows users to register custom metrics, processors, or analysis functions while maintaining system stability and security through well-defined interfaces and resource limitations.

Metric Registry: A centralized system for metric discovery, registration, and management that provides a unified interface for accessing both built-in and custom metrics while handling version compatibility and dependency management.

Circuit Breaker: A failure isolation pattern that monitors the health of external dependencies (like LLM APIs) and automatically stops sending requests when failure rates exceed acceptable thresholds, preventing cascade failures and resource exhaustion.

Backpressure Handling: Mechanisms for preventing fast-producing components from overwhelming slower consumers by implementing flow control, buffering strategies, and graceful degradation when system capacity is exceeded.

Event-Driven Architecture: A system design approach where components communicate through asynchronous event notifications rather than direct method calls, enabling loose coupling, better scalability, and more resilient failure handling.

Distributed Tracing: The practice of tracking requests and operations across component boundaries using correlation IDs and structured logging, enabling diagnosis of complex issues that span multiple system components.

Error Handling and Recovery

Error Boundaries: Isolation mechanisms that contain failures within specific components or operations, preventing errors from cascading through the entire system and enabling partial functionality during degraded conditions.

Graceful Degradation: System behavior that maintains partial functionality when some components or capabilities are unavailable, such as falling back to simpler metrics when advanced evaluators fail or using cached results when APIs are unreachable.

Exponential Backoff: A retry strategy where the delay between retry attempts increases exponentially with each failure, helping to avoid overwhelming already-stressed systems while still providing reasonable recovery times.

Recovery Strategies: Systematic approaches for handling different categories of failures, including automatic retry with backoff, failover to alternative providers, degraded mode operation, and manual intervention escalation paths.

State Reconstruction: The process of rebuilding consistent system state after data corruption or partial failures, using techniques like write-ahead logs, checksums, and redundant storage to ensure data integrity.

Thundering Herd: A failure pattern where multiple processes simultaneously attempt to access a recovered service, potentially overwhelming it again. Prevention strategies include jittered delays, circuit breakers, and gradual traffic ramping.

Quality Assurance and Testing

Test Pyramid: A testing strategy that emphasizes fast, isolated unit tests at the base, integration tests for component interactions in the middle, and comprehensive end-to-end tests at the apex, balancing thorough coverage with execution speed.

Milestone Checkpoints: Specific verification criteria and success indicators defined for each development milestone, providing clear progress markers and quality gates throughout the implementation process.

Mock Providers: Test implementations of external dependencies (like LLM APIs) that provide predictable, controllable responses for isolated testing without incurring costs or depending on external service availability.

Regression Testing: Systematic testing to ensure that changes don't break existing functionality, particularly important for evaluation systems where subtle changes can significantly impact result interpretation.

Test Data Factory: Utilities for generating realistic test data with consistent characteristics, enabling reproducible testing across different scenarios while maintaining data quality and representativeness.

Debugging and Diagnostics

Symptom Classification: The systematic categorization of evaluation issues based on observable behaviors, enabling efficient diagnosis by narrowing the potential root causes before beginning detailed investigation.

Component Isolation: Debugging techniques that systematically narrow down which system component is responsible for observed issues, using controlled testing and dependency injection to isolate problematic areas.

Correlation ID Tracking: The practice of assigning unique identifiers to requests and operations that persist across all log entries and system interactions, enabling complete trace reconstruction for debugging complex distributed operations.

Structured Logging: A logging approach that produces machine-readable, consistently formatted log entries with standardized fields, enabling automated analysis and correlation of log data across system components.

State Inspection: Debugging capabilities that allow real-time examination of internal component state, including data structures, configuration values, cache contents, and process status, without requiring system restart.

Cache Validation: Systematic verification of cached data integrity and consistency, including detection of corruption, staleness, and key collision issues that can cause incorrect evaluation results.

Advanced and Future Concepts

Multi-Modal Evaluation: Assessment methodologies that handle multiple data types (text, images, audio, video) within the same evaluation framework, requiring specialized processors and metrics that can reason across modalities.

Adaptive Evaluation Strategies: Dynamic approaches that adjust evaluation depth and complexity based on test case characteristics, initial results, or resource constraints, optimizing the trade-off between evaluation thoroughness and computational cost.

Hierarchical Evaluation Taxonomies: Organizational structures that represent evaluation capabilities as skill trees with dependencies, enabling fine-grained assessment of specific model capabilities and identification of capability gaps.

Distributed Evaluation Infrastructure: Architectural approaches for scaling evaluation across multiple machines or data centers, including work distribution, result aggregation, and consistency management in distributed environments.

MLOps Pipeline Integration: Interfaces and protocols for embedding evaluation capabilities into broader machine learning operations workflows, including automated quality gates, continuous evaluation, and deployment decision automation.

Evaluation-Guided Experimentation: Methodologies that use offline evaluation results to inform online A/B testing design, hypothesis generation, and resource allocation for model improvement efforts.

Statistical and Mathematical Concepts

Confidence Intervals: Statistical ranges that provide uncertainty bounds around evaluation metrics, helping interpret whether observed differences are within expected variation or represent genuine performance changes.

Statistical Power: The probability that a statistical test will correctly identify a true effect when one exists, important for designing evaluation studies with sufficient sample sizes to detect meaningful performance differences.

Outlier Detection: Techniques for identifying evaluation results that fall far outside expected ranges, which may indicate data quality issues, model failures, or exceptional cases that deserve special attention.

Distribution Analysis: Statistical examination of how evaluation scores are distributed across test cases, helping identify skewness, multimodality, or other patterns that inform interpretation of aggregate statistics.

Temporal Analysis: Examination of how evaluation metrics change over time, including trend analysis, seasonality detection, and identification of sudden shifts in performance characteristics.

Operational and Maintenance Concepts

Health Monitoring: Continuous assessment of system component status, performance metrics, and resource utilization to enable proactive identification and resolution of operational issues.

Capacity Planning: Analysis and forecasting of computational resource requirements based on evaluation workload characteristics, growth projections, and performance targets.

Data Lineage: Tracking and documentation of how evaluation datasets evolve over time, including source attribution, transformation history, and dependency relationships between different dataset versions.

Audit Trail: Comprehensive logging of all evaluation activities, configuration changes, and administrative actions to support compliance requirements, debugging, and performance analysis.

Performance Profiling: Systematic measurement and analysis of system resource usage, bottlenecks, and optimization opportunities to maintain evaluation system efficiency as workloads scale.

Implementation Guidance

Understanding these terms is crucial for implementing a robust LLM evaluation framework, but terminology alone isn't sufficient. The concepts must be properly operationalized through careful system design and implementation.

Technology Recommendations for Terminology Management

Component	Simple Option	Advanced Option
Documentation	Markdown files with glossary	Sphinx with automated term linking
Term Validation	Manual review during PR process	Automated terminology checking in CI/CD
Cross-References	Manual markdown links	Automated cross-reference generation
Version Control	Git for documentation	Specialized terminology management tools

Recommended Documentation Structure

When implementing the evaluation framework, organize terminology documentation to support both learning and operational use:

```
docs/
  glossary/
    core-concepts.md      ← Fundamental evaluation terminology
    dataset-management.md ← Dataset-specific terms
    metrics-scoring.md   ← Metrics and scoring terminology
    execution-processing.md ← Runtime and processing terms
    analysis-reporting.md ← Statistical and reporting terms
    system-architecture.md ← Infrastructure terminology
    cross-references.md   ← Term relationships and dependencies
  examples/
    terminology-usage.md ← Concrete examples of term usage
    common-mistakes.md    ← Frequently misused terms
```

Terminology Validation Infrastructure

Implement systematic validation to ensure consistent terminology usage across the codebase:

```
# terminology_validator.py - Complete infrastructure starter code
```

PYTHON

```
import re

from pathlib import Path

from typing import Dict, List, Set, Optional

from dataclasses import dataclass


@dataclass

class TermDefinition:

    """Definition of a standardized term with validation rules."""

    term: str

    definition: str

    category: str

    aliases: List[str]

    deprecated_alternatives: List[str]

    usage_examples: List[str]


class TerminologyValidator:

    """Validates consistent terminology usage across documentation and code."""

    def __init__(self, glossary_path: Path):

        self.glossary_path = glossary_path

        self.standard_terms: Dict[str, TermDefinition] = {}

        self.load_standard_terminology()

    def load_standard_terminology(self) -> None:

        """Load standard terminology definitions from glossary files."""

        # TODO: Implement glossary parsing from markdown files

        # TODO: Extract term definitions, aliases, and deprecated alternatives

        # TODO: Build lookup indexes for efficient validation

        # TODO: Validate glossary consistency (no conflicting definitions)

        pass
```

```
def validate_document(self, doc_path: Path) -> List[str]:  
  
    """Validate terminology usage in a documentation file."""  
  
    violations = []  
  
  
    # TODO: Read document content and extract terminology usage  
  
    # TODO: Check for deprecated terms and suggest replacements  
  
    # TODO: Verify that technical terms are properly defined  
  
    # TODO: Ensure consistent capitalization and spelling  
  
    # TODO: Check for undefined terms that should be in glossary  
  
  
    return violations  
  
  
def validate_code_comments(self, code_path: Path) -> List[str]:  
  
    """Validate terminology in code comments and docstrings."""  
  
    violations = []  
  
  
    # TODO: Extract comments and docstrings from source code  
  
    # TODO: Apply same validation rules as documentation  
  
    # TODO: Check that API documentation uses standard terms  
  
    # TODO: Verify that error messages use consistent terminology  
  
  
    return violations  
  
  
def generate_term_usage_report(self, project_root: Path) -> Dict[str, int]:  
  
    """Generate report of terminology usage frequency across project."""  
  
    # TODO: Scan all documentation and code files  
  
    # TODO: Count usage frequency for each standard term  
  
    # TODO: Identify unused terms that might be candidates for removal
```

```
# TODO: Find commonly used terms not in the standard glossary

    return {}

# Automated terminology checking for CI/CD pipeline

def check_terminology_compliance(project_root: Path) -> bool:

    """Check project-wide terminology compliance for CI/CD integration."""

    validator = TerminologyValidator(project_root / "docs" / "glossary")

    violations = []

    # Check all documentation files

    for doc_file in (project_root / "docs").rglob("*.md"):

        violations.extend(validator.validate_document(doc_file))

    # Check code comments and docstrings

    for code_file in project_root.rglob("*.py"):

        violations.extend(validator.validate_code_comments(code_file))

    if violations:

        print("Terminology violations found:")

        for violation in violations:

            print(f"  - {violation}")

        return False

    return True
```

Core Logic Skeleton for Term Management

For teams implementing glossary management as part of their evaluation framework:

```
class GlossaryManager:

    """Manages glossary definitions and cross-references for evaluation framework."""

    def __init__(self, storage_path: Path):
        self.storage_path = storage_path
        self.terms: Dict[str, TermDefinition] = {}
        self.cross_references: Dict[str, Set[str]] = {}

    def add_term_definition(self, term: str, definition: str, category: str) -> None:
        """Add a new term definition to the glossary."""

        # TODO: Validate that term doesn't already exist with different definition
        # TODO: Extract cross-references to other terms mentioned in definition
        # TODO: Update bidirectional cross-reference mappings
        # TODO: Persist updated glossary to storage
        # TODO: Validate definition completeness and clarity
        pass

    def find_related_terms(self, term: str) -> List[str]:
        """Find terms related to the given term through cross-references."""

        # TODO: Use cross-reference graph to find directly related terms
        # TODO: Include terms that share the same category
        # TODO: Find terms mentioned in the definition
        # TODO: Return ordered list by relevance score
        pass

    def validate_definition_completeness(self, term: str) -> List[str]:
        """Validate that a term definition meets completeness criteria."""

        issues = []

        # TODO: Check that definition exists and is non-empty
```

```

# TODO: Verify that definition doesn't use undefined terms

# TODO: Ensure definition provides sufficient context

# TODO: Check for circular definitions

# TODO: Validate that examples are helpful and accurate


return issues

def generate_glossary_html(self) -> str:

    """Generate interactive HTML glossary with cross-references."""

    # TODO: Create HTML template with term navigation

    # TODO: Add cross-reference links between related terms

    # TODO: Include search functionality for finding terms

    # TODO: Organize terms by category with filtering

    # TODO: Add examples and usage guidance for each term

    pass

```

Language-Specific Implementation Hints

When implementing terminology management in Python:

- Use `dataclasses` for term definitions to get automatic equality and serialization
- Use `pathlib.Path` for cross-platform file handling in documentation processing
- Use `re` module for extracting terminology patterns from text
- Use `typing` annotations extensively for clear interfaces
- Use `json` or `yaml` for persisting glossary data with version control
- Use `sphinx` for advanced documentation generation with term linking

Milestone Checkpoint: Terminology Consistency

After implementing terminology management:

Validation Commands:

```

python terminology_validator.py --check-project .

python -m pytest tests/test_terminology.py -v

```

BASH

Expected Behavior:

- All standard terms should be consistently used across documentation

- No deprecated terminology should appear in new code or documentation
- Cross-references between related terms should be automatically maintained
- Glossary should be searchable and well-organized by category

Signs of Issues:

- Inconsistent capitalization of technical terms across files
- Use of deprecated alternatives instead of standard terminology
- Missing definitions for commonly used technical terms
- Broken cross-references in generated documentation

Debugging Terminology Issues

Symptom	Likely Cause	Diagnosis	Fix
Term used inconsistently	No standard definition	Search codebase for variations	Add to standard glossary
Cross-references broken	Definition moved or renamed	Check glossary update history	Update all references
Glossary feels incomplete	Terms used but not defined	Run terminology extraction	Add missing definitions
Definitions are circular	Poor definition structure	Analyze definition dependencies	Rewrite with clear hierarchy
Users confused by terms	Definitions too technical	Review with target audience	Simplify with examples

The glossary serves as the foundation for clear communication throughout the LLM evaluation framework implementation. By establishing precise, consistent terminology, teams can avoid the confusion and misalignment that often derail complex technical projects. The investment in terminology management pays dividends through improved code quality, better documentation, and more effective collaboration.