README

HuleEdu Microservice Platform

HuleEdu is an educational technology platform for automated essay processing and assessment. It is built as a collection of specialized microservices orchestrated into a unified workflow for tasks such as file ingestion, text content storage, spell checking, and AI-based comparative judgment of student essays.

The system has been rebuilt from a legacy monolithic application into a modern event-driven microservice architecture to improve scalability, maintainability, and clear separation of concerns. All services and shared components reside in a single monorepo for synchronized development.

This document provides a technical overview of the system architecture, usage guidelines for developers, development standards, current implementation status, and planned future enhancements.

Architecture and Design

Core Architectural Principles

Domain-Driven Design (DDD)

The platform is divided into services by bounded context. Each microservice owns its domain logic and data store. Service boundaries are strictly enforced (no direct database access across services, no tightly coupled logic).

Event-Driven Architecture (EDA)

Microservices communicate primarily through asynchronous events via a Kafka message bus. Cross-service interactions are decoupled using Kafka topics (with a standardized event schema), minimizing the need for direct synchronous calls between services.

Explicit Data Contracts

All inter-service communication models (event payloads, API request/response schemas) are defined as versioned Pydantic models in a shared `common_core` library. A standardized `EventEnvelope` wrapper is used for all Kafka events to provide metadata (timestamps, origin, schema version, correlation IDs, etc.) and ensure compatibility across services.

Service Autonomy

Each service is independently deployable and has its own data persistence. One service will never directly query or write to another service's database; any data sharing occurs via published events or well-defined internal APIs. This autonomy allows services to scale and evolve in isolation.

Asynchronous I/O

All services are written using Python's `async`/`await` and asynchronous frameworks:

- **Web Services**: Quart or FastAPI
- **Kafka Clients**: aiokafka
- **Database Access**: async SQLAlchemy

Non-blocking I/O ensures that each service can handle high concurrency efficiently.

Dependency Injection (DI)

The codebase employs a custom DI framework (Dishka) to invert dependencies and facilitate testing. Each service defines abstract interfaces (`typing.Protocol` classes in a `protocols.py`) for its key operations or external interactions. Concrete implementations are provided and wired at runtime via a DI container (see each service's `di.py`). This ensures business logic depends on interfaces, making components swappable and modular.

Configurable via Environment

Services use Pydantic `BaseSettings` classes (in each service's `config.py`) to load configuration from environment variables (with support for `.env` files in development). This centralizes configuration (e.g. database URLs, API keys, service ports) and makes services twelve-factor compliant. No configuration values are hard-coded; all are injected via environment or configuration files.

Centralized Logging & Observability

A shared logging utility (`huleedu_service_libs.logging_utils`) provides structured logging for all services. Logs include correlation IDs flowing through event metadata and request headers, enabling traceability across service boundaries.

Each service avoids using the standard logging module directly and instead uses the centralized logger to ensure a uniform format. In addition, every service exposes:

- `/healthz` endpoint for health checks
- `/metrics` endpoint for Prometheus scraping

This contributes to a consistent observability stack.

Monorepo Structure

The repository is organized as a monorepo managed by PDM (Python Dependency Manager), containing all services and shared code in one place for easy coordination:

- **`common_core/`** – Shared Python package defining common data models and enums used across services. This includes Pydantic models for events and API DTOs (Data Transfer Objects), standardized enumerations (e.g., for statuses, error codes), and the base event envelope format. This library is the source of truth for inter-service data contracts.

- **`services/`** – Directory holding all microservices, each in its own sub-folder. For example:

  - `services/content_service/`
  - `services/spellchecker_service/`
  - `services/batch_orchestrator_service/`

  (See Microservices Overview below for details on each service)

- **`services/libs/`** – Shared service libraries (internal utility packages). These include common infrastructure code such as Kafka client wrappers, Redis clients, and logging/monitoring helpers that are used by multiple services.

- **`scripts/`** – Utility and setup scripts for development and operations. Notable scripts include:
    - `setup_huledu_environment.sh` – Bootstraps the development environment (installs PDM if missing, then installs all packages in the monorepo).
    - `kafka_topic_bootstrap.py` – Script to create all required Kafka topics (run automatically on startup in Docker Compose).
    - Other convenience scripts for Docker orchestration and testing (e.g., `docker-rebuild.sh`, `validate_batch_coordination.sh`).
- **`documentation/`** – Design and planning documents. This includes product requirement docs (PRDs), architecture decision records, and task breakdowns for major development phases. For example, the `SERVICE_FUTURE_ENHANCEMENTS/` and `TASKS/` subfolders contain specs and implementation notes for new features and phases of the project.
- **`.windsurf/rules/`** – The repository's development standards and rules in machine-readable Markdown format. Each rule file (e.g., coding standards, service architecture requirements, testing practices) is kept here. The master index `000-rule-index.mdc` lists all rules. These rules are used to ensure consistency and quality across the codebase (often enforced via review or tooling).

Microservices Overview

The HuleEdu platform is composed of multiple microservices, each responsible for a specific aspect of the overall system. All services are implemented in Python (>=3.11) and use asynchronous frameworks.

They communicate via Kafka events and occasional internal HTTP calls. Below is a brief overview of each service and its role:

Content Service

A Quart-based HTTP service for binary content storage and retrieval. It handles storing essay files or text in a filesystem-based repository (local disk for development; could be S3 or similar in production). It exposes a simple REST API on port 8001 (e.g., `POST/GET /v1/content` for uploading or fetching content by ID). Other services (like File Service and Essay Lifecycle) use this service to persist and retrieve raw text or file contents.

File Service

A Quart-based HTTP service (port 7001) that handles file uploads and content ingestion workflow. This service accepts multipart file uploads (for example, a batch of student essay files) via an endpoint such as `POST /v1/files/batch`. For each uploaded file, it extracts text content (performing PDF/DOCX text extraction if necessary), stores the content via the Content Service, and emits an `EssayContentProvisioned` event to signal that an essay's text is ready. It acts as the entry point for new essay data entering the system, orchestrating with Essay Lifecycle Service to initialize essay records. The File Service also publishes events to Kafka to trigger downstream processing (for example, notifying that a new batch of essays is ready for spell checking).

Essay Lifecycle Service (ELS)

A hybrid service with both an HTTP API (port 6001) and a Kafka event consumer component. ELS maintains the state of each essay through the processing pipeline using a formal state machine. It defines an `EssayStateMachine` (via the `transitions` library) that governs allowed state transitions (e.g., from `content_provisioned` to `spellcheck_completed` to `cj_completed`, etc.).

The service receives commands and events to advance essay state: for example, it consumes a `BatchSpellcheckInitiateCommand` event to start spell checking all essays in a batch, and later emits an `EssaySpellcheckCompleted` event when an essay's spellcheck phase is done. ELS also exposes HTTP endpoints for querying or updating essay state if needed (and for internal coordination with Batch Orchestrator).

For persistence, it uses a relational database to store essay metadata and state (SQLite in development, PostgreSQL in production), accessed via SQLAlchemy. ELS ensures exactly-once state transitions and acts as the source of truth for an essay's progress through the pipeline.

Batch Orchestrator Service (BOS)

A Quart-based HTTP service (port 5001) that coordinates processing at the batch level. A "batch" represents a collection of essays (e.g., a class assignment submission batch) to be processed together. BOS provides APIs to register a new batch and to initiate processing on a batch (triggering the pipeline of analysis phases for all essays in that batch).

When a client (or the API Gateway) requests a batch pipeline execution via BOS, the BOS will consult the Batch Conductor Service to determine the correct sequence of phases based on the current state of the batch and system configuration. BOS then issues commands to the Essay Lifecycle Service (e.g., instruct ELS to start spellchecking all essays in the batch) and listens for phase completion events on Kafka. In essence, BOS is the central orchestrator that drives the multi-phase essay analysis workflow, orchestrating between ELS and various processing services.

BOS does not perform heavy processing itself; it issues commands and reacts to events, maintaining batch-level context and progress.

Batch Conductor Service (BCS)

An internal orchestration logic service (Quart-based, port 4002 for its API). BCS's responsibility is dynamic pipeline dependency resolution. BOS delegates to BCS when it needs to determine which phase of processing should happen next for a given batch.

BCS keeps track of what processing has been completed for each batch by consuming all relevant events (e.g., it listens to essay-level completion events from ELS and results from analysis services). Using this information, BCS computes whether the prerequisites for the next phase are satisfied. For example, if the pipeline is `Spell Checking -> Comparative Judgment`, BCS will ensure all essays in the batch have spellcheck results before allowing the BOS to trigger the Comparative Judgment phase.

BCS uses Redis as an in-memory store to manage batch state and coordinate complex transitions (employing atomic operations and optimistic locking via Redis transactions to avoid race conditions in concurrent event processing). It also provides an API (POST /internal/v1/pipelines/define) for BOS to request a pipeline resolution (this API returns the next phase or indicates completion). BCS implements robust error handling: if it detects an inconsistency or failure in phase progression, it can push a message to a Dead Letter Queue (DLQ) topic for later analysis. In summary, BCS adds intelligence to the orchestration process, enabling dynamic pipelines that adapt to real-time results and conditions.

Spellchecker Service

A Kafka consumer microservice (no public HTTP API) dedicated to spelling and grammar analysis of essay text. This service listens on a Kafka topic (e.g. `huleedu.commands.spellcheck.v1`) for commands to spell-check a particular essay. Upon receiving a command, it retrieves the essay text (from Content Service or included in the event payload), then performs spell checking and linguistic error analysis. It incorporates both standard spell-checking (via libraries like `pyspellchecker`) and second-language (L2) error correction logic for non-native writing issues. After processing, it emits an `EssaySpellcheckCompleted` event containing the results (e.g. lists of errors found and corrections). The Spellchecker Service runs as an asynchronous worker and typically handles many essays in parallel from the event queue. It also exposes a Prometheus metrics endpoint (on a small HTTP server at port 8002) to report its operation status (e.g. number of essays processed, processing duration, etc.). This service is a key part of the first phase in the essay processing pipeline.

Key Features:

- **Event-Driven**: Listens on Kafka topic `huleedu.commands.spellcheck.v1`
- **Language Support**:
  - Standard spell-checking (using libraries like `pyspellchecker`)
  - Second-language (L2) error correction for non-native writing
- **Asynchronous Processing**: Handles multiple essays in parallel
- **Monitoring**: Exposes Prometheus metrics endpoint on port 8002

Workflow:

- Receives spell-check command via Kafka
- Retrieves essay text from Content Service or event payload
- Performs linguistic analysis
- Emits `EssaySpellcheckCompleted` event with results

Comparative Judgment (CJ) Assessment Service

A Kafka-driven service (with optional HTTP endpoints for health checks on port 9095) that performs AI-assisted comparative judgment of essays. In comparative judgment, essays are evaluated by comparing them in pairs. This service uses Large Language Model (LLM) APIs to generate pairwise comparisons or scores between essays in a batch. It listens on a Kafka topic (e.g. `huleedu.commands.cj_assess.v1`) for commands to assess a batch or a pair of essays. Internally, the CJ service interacts with the LLM Provider Service (described below) to obtain AI-generated judgments in a resilient way. It may break a large task (ranking a whole batch of essays) into many pairwise comparison queries to the LLM provider. The service collates the results (e.g. which essays won comparisons) and from these produces a ranked list or scores for all essays in the batch. Once comparative assessment for the batch is complete, it emits a `BatchComparativeJudgmentCompleted` event with the outcome (e.g. relative rankings or scores for each essay). This event can then be used by other components (Result Aggregator or BOS) to finalize the batch results. The CJ Assessment Service uses a PostgreSQL database to store intermediate results and ensure consistency (especially since LLM calls may be slow or need retries). Metrics and health endpoints are available (on `/metrics` and `/healthz`) for monitoring. This service enables automated scoring or ranking of essays using AI, providing the core of the grading or feedback mechanism.

Key Features:

- **Port**: 9095 (HTTP endpoints for health checks)
- **Event-Driven**: Listens on Kafka topic `huleedu.commands.cj_assess.v1`
- **AI Integration**: Uses LLM Provider Service for pairwise comparisons
- **Data Persistence**: PostgreSQL database for storing intermediate results
- **Monitoring**:
  - `/metrics` endpoint for Prometheus
  - `/healthz` endpoint for health checks

Workflow:

- Receives assessment command via Kafka
- Breaks down batch assessment into pairwise comparisons
- Uses LLM Provider Service for AI judgments
- Collates results into ranked list or scores
- Emits `BatchComparativeJudgmentCompleted` event with outcomes

LLM Provider Service

A specialized Quart-based HTTP service (port 8090) that acts as a gateway and queue for calls to external Large Language Model providers. Multiple services in the platform (especially the CJ Assessment and future AI-driven services) need to call external AI APIs (like OpenAI, Anthropic, etc.). Instead of each service handling these calls (which can be slow or have rate limits), the LLM Provider Service centralizes this function. It exposes endpoints such as `POST /api/v1/comparison` to submit a comparison or other AI request. Requests are queued (in Redis) and processed asynchronously to handle high load and avoid hitting external API limits. The service implements circuit breakers and fallback strategies: if one AI provider is down or returns errors, it can switch to an alternative provider or degrade gracefully. Clients (like the CJ service) receive an immediate acknowledgment (HTTP 202 Accepted with a queue ID) and can poll `/api/v1/status/{queue_id}` or `/api/v1/results/{queue_id}` to get the result once ready. This design gives resilience to the AI calls and decouples the rest of the system from external API latency or failures.

The LLM Provider Service supports multiple AI providers (OpenAI, Anthropic, Google PaLM, OpenRouter, etc.) through a unified interface, and does no caching of responses (each request passes through to preserve the psychometric validity of CJ assessments). It uses Redis both for queuing requests and as a short-term results store. This service is critical for any AI-driven feature in HuleEdu, ensuring those features are robust and scalable.

Key Features:

- **Centralized AI API Management**:
  - Handles rate limiting
  - Implements circuit breakers
  - Provides fallback strategies
- **Asynchronous Processing**: Uses Redis for request queuing
- **Multi-Provider Support**: Works with OpenAI, Anthropic, and other LLM providers

API Endpoints:

- `POST /api/v1/comparison` - Submit AI comparison request
- `GET /api/v1/status/{queue_id}` - Check request status
- `GET /api/v1/results/{queue_id}` - Retrieve results

Resilience Features:

- Automatic retries for failed requests
- Load balancing across multiple AI providers
- Graceful degradation when providers are unavailable

Class Management Service (CMS)

A Quart-based HTTP CRUD service (port 5002) that manages metadata about classes, students, and their enrollment relationships. It serves as the authoritative source for user domain data: which classes exist, which students belong to which class, etc. Other services (like the API Gateway or any feature that needs student info) rely on CMS for querying class/student info. It provides REST endpoints under `/v1/classes` and `/v1/students` for creating, reading, updating, and deleting these records. For instance, an admin could create a new class and assign students to it via this service's API. The service uses a PostgreSQL database to persist class and student information. All access to class/student data from other parts of the system goes through this service (directly or via the API Gateway), ensuring a single source of truth. This microservice illustrates the platform's domain separation: educational administrative data is handled separately from essay processing data.

Key Features:

- **Authoritative Source**: Serves as the single source of truth for user domain data
- **Data Management**:
  - Tracks which classes exist
  - Manages student enrollments
- **REST API**:
  - `/v1/classes` - Manage class records
  - `/v1/students` - Manage student records
- **Database**: PostgreSQL for persistent storage

Result Aggregator Service (RAS)

A hybrid service (Kafka consumer with an internal HTTP interface on port 4003) responsible for aggregating and materializing the results of essay processing for fast retrieval.

Key Features:

- **Event Processing**: Listens to all completion events on Kafka
- **Data Storage**: Maintains materialized views in PostgreSQL
- **API Endpoint**: `GET /internal/v1/batches/{batch_id}/status` - Returns comprehensive status and results
- **CQRS Pattern**: Separates command (write) and query (read) operations
- **Performance**: Uses Redis caching for frequent queries
- **Security**: Implements service-to-service authentication

API Gateway Service

A FastAPI-based gateway service (port 4001) that serves as the unified entry point for external clients (e.g., a Svelte frontend or third-party applications).

Core Responsibilities:

- **Authentication**: Validates JWT tokens for incoming requests
- **Request Validation**: Uses Pydantic models from `common_core`
- **Rate Limiting**: Protects backend services from excessive traffic
- **Request Routing**: Proxies requests to appropriate internal services
- **Event Publishing**: Publishes client requests as Kafka events
- **WebSocket Support**: Enables real-time updates for clients
- **Anti-Corruption Layer**: Translates between internal and client-facing data models

WebSocket Service

A dedicated service for managing persistent WebSocket connections with clients. It works in conjunction with the API Gateway to provide real-time updates on essay processing status.

Key Features:

- **Real-Time Updates**: Pushes status changes to clients as they happen.
- **Scalability**: Designed to handle a large number of concurrent connections.
- **Integration**: Listens to internal Kafka events to know when to notify clients.

Technology Stack

HuleEdu leverages a modern Python-based tech stack and tooling:

Core Technologies

Python 3.11+

- Primary programming language for all services
- Chosen for its rich ecosystem and async support

Web Frameworks

- **Quart**: ASGI-compatible Flask variant used for most HTTP services
- **FastAPI**: Used for API Gateway and lightweight APIs
  - High performance
  - Built-in data validation
  - Async route handlers
  - Automatic OpenAPI documentation

Data Validation & Serialization

- **Pydantic**:
  - Defines schemas for configuration and data models
  - Validates all request/response bodies
  - Ensures data consistency across services
  - Used for Kafka event payloads

Event Streaming

- **Apache Kafka**:
  - Backbone of event-driven architecture
  - Uses `aiokafka` for Python clients
  - Provides scalable, durable message queuing
  - Enables asynchronous workflows
  - Supports event replay and ordering

Database Solutions

PostgreSQL

- Primary production database
- Used for services requiring robust data storage

- Handles Class Management, Result Aggregator, and CJ service data

SQLite

- Default for development environments
- Simplifies local development setup

ORM & Database Access

SQLAlchemy

- Async ORM for database operations
- Provides abstraction layer over SQL
- Enables database-agnostic code
- Supports migrations and schema management

Redis

In-memory data store used for caching and transient data coordination.

Key Use Cases:

- **Batch Coordination**: Maintains batch state and critical section locks (using WATCH/MULTI transactions)
- **Request Queuing**: Used by LLM Provider and API Gateway
- **Rate Limiting**: Enforces request rate limits
- **Caching**: Speeds up frequent queries (e.g., in RAS)
- Atomic operations for data consistency
- Pub/sub capabilities for event-driven patterns
- Low-latency performance
- Built-in concurrency control

Dishka (Dependency Injection)

Custom DI framework integrated with Quart (`quart_dishka`) for managing service components.

Key Features:

- **Loose Coupling**: Binds implementations to interfaces at runtime.
- **Test-Friendly**: Simplifies testing by allowing easy swapping of implementations.
- **Clean Architecture**: Supports clean architecture patterns through dependency inversion.

Usage Example:

```python
# Service definition
class DatabaseService(Protocol):
    def get_data(self) -> Data: ...

# Production implementation
class PostgresDatabaseService(DatabaseService):
    def get_data(self) -> Data:
        # Implementation using PostgreSQL
        pass

# Test implementation
class MockDatabaseService(DatabaseService):
    def get_data(self) -> Data:
        # Mock implementation for testing
        return TestData()
```

Containerization & Orchestration

Docker & Docker Compose

Key Features:

- Containerization of all services and dependencies
- Consistent runtime environments
- Simplified local development and testing
- Production-parity in development

Components:

- **Dockerfiles**: One per microservice
- **docker-compose.yml**: Central configuration for all services
- **Dependencies**:
  - Kafka
  - Zookeeper
  - Redis
  - PostgreSQL

Development Tools

PDM (Python Dependency Manager)

PDM is used to manage Python packages in our monorepo. It provides several key benefits:

- **Editable Packages**: Each service can be installed as an editable package
- **Unified Lockfile**: Single source of truth for all dependencies
- **Task Runner**: Built-in support for common development tasks
- **Modern Workflow**: Replaces traditional tools like pip/venv and Makefiles

Common Commands:

```
# Install dependencies
pdm install

# Run tests
pdm run test

# Run linter
pdm run lint

# Run formatter
pdm run format
```

Ruff (Linter & Formatter)

Ruff is a fast Python linter and code formatter configured for the project. It enforces coding style (PEP8 compliance, import sorting, etc.) and can automatically apply simple formatting fixes.

The project uses Ruff to:

- Flag style and syntax issues
- Automatically format code (via `pdm run format-all`)
- Enforce consistent code style across the codebase
- Speed up code reviews by catching issues early

MyPy (Static Type Checker)

MyPy is used throughout the codebase to ensure type safety and catch potential issues at development time.

Key Benefits:

- Full type hinting support
- Catches type errors before runtime
- Ensures interface contracts are maintained
- Improves code quality in a large codebase
- Works well with protocols and dependency injection

PyTest (Testing Framework)

PyTest is the testing framework used for all automated tests in the project.

Testing Strategy:

- **Unit Tests**: Test individual components in isolation
- **Integration Tests**: Verify interactions between components
- **Contract Tests**: Ensure events and APIs conform to expected schemas
- **End-to-End Tests**: Test complete workflows

Features:

- Run tests via `pdm run test-all`
- Support for testing with real dependencies
- Temporary database and Kafka instances for testing
- Extensive use of fixtures for test setup
- Plugins for async testing and code coverage

Development Setup and Usage

This section provides instructions for developers who want to set up, run, or extend the system locally.

Prerequisites

- **Python 3.11 or above** - Required for running the services and tools
- **Docker with Docker Compose** - Required for running dependencies in containers
- **PDM** - Python Dependency Manager (will be installed automatically if missing)

Initial Environment Setup

1. Clone the repository to your local machine:

   ```
   git clone <repository-url>
   cd huledu-reboot
   ```

2. Run the setup script to install dependencies:

   ```
   ./scripts/setup_huledu_environment.sh
   ```

   This script will:

   - Install PDM if not already present
   - Set up a virtual environment (`.venv`)
   - Install all Python dependencies
   - Register each service package in development mode
   - Configure pre-commit hooks

     > ***Note:*** *The project uses PDM instead of pip or Poetry. Do not use `pip install .` directly. The setup script and `pdm install` ensure the correct environment. All commands below assume the PDM-managed virtual environment is active (the setup script activates it automatically, or you can run `pdm shell` manually).*

Running the Full System with Docker Compose

The recommended way to run all microservices together (along with required infrastructure like Kafka) is using Docker Compose. PDM provides helper scripts to simplify this process.

Building Service Images

From the project root, build the Docker images for all microservices:

```
pdm run docker-build
```

This command will:

1. Build all service images in parallel

2. Tag them with the current branch name and `latest`
3. Cache intermediate layers for faster subsequent builds

This will use the Dockerfiles in each service directory to create images tagged for this project. It may take some time on first build as it installs OS packages and Python dependencies inside the images.

Launching Services with Docker Compose

To start all services, run:

```
pdm run docker-up
```

This command starts the Docker Compose stack defined in `docker-compose.yml`. It will spin up:

Core Services:

- Zookeeper and Kafka brokers (for the event bus)
- Redis (for BCS state and LLM queue)
- PostgreSQL (with separate databases for services)
- Kafka topic initializer container (runs once at startup)

HuleEdu Microservices:

- Content Service
- File Service
- ELS (Event Logging Service)
- BOS (Batch Orchestration Service)
- BCS (Batch Coordination Service)
- Spellchecker Service
- CJ (Comparative Judgment) Service
- LLM Provider Service
- CMS (Class Management Service)
- RAS (Result Aggregator Service)
- API Gateway
- And other supporting services

Verifying the Deployment

Once started, services run in the background (detached mode). You can verify they're running by:

1. Checking container status:

```
docker-compose ps
```

2. Accessing health endpoints:
   - BOS: `http://localhost:5001/healthz`
   - API Gateway: `http://localhost:8000/health`
   - (Ports may vary based on configuration)

Interacting with the System

Once all containers are up, you can:

1. **Upload Files**:
   - Use the File Service API
   - Or go through the API Gateway
2. **Process Batches**:
   - Trigger batch processing via the BOS API
3. **Monitor Activity**:
   - Observe logs and events
   - Check service metrics and health

For development and testing, you can use the provided scripts or write unit/integration tests. See individual service README files for specific API documentation and examples.

Shutting Down

To stop all services:

```
pdm run docker-down
```

This will:

- Stop and remove all containers
- Preserve data in volumes (PostgreSQL, file storage, etc.)
- Maintain configuration between restarts

Additional Commands

- **View Logs**:

  ```
  pdm run docker-logs
  ```

- **Restart Services**:

```
pdm run docker-restart
```

- **Remove Volumes** (use with caution):

```
docker-compose down -v
```

Note: The `docker-restart` command combines `docker-down` and `docker-up`, which is useful when you've changed code and rebuilt images.

Common Development Tasks

All routine development tasks are encapsulated in PDM scripts (see the `[tool.pdm.scripts]` section of `pyproject.toml`). Here are key commands to be run from the project root:

Code Quality

- **Format Code**: `pdm run format-all` - Auto-format the codebase using Ruff's formatting rules
- **Lint Code**: `pdm run lint-all` - Run the linter (Ruff) on all files
- **Fix Lint Issues**: `pdm run lint-fix` - Automatically fix lint issues where possible
- **Type Checking**: `pdm run typecheck-all` - Execute MyPy across the monorepo

Testing

- **Run All Tests**: `pdm run test-all` - Execute all tests for all services
- **Parallel Testing**: `pdm run test-parallel` - Force parallel execution (default)
- **Sequential Testing**: `pdm run test-sequential` - Run tests serially when needed

The test suite includes:

- Unit tests for each service
- Integration tests involving multiple services
- Contract tests for shared models

Docker Workflow

In addition to `docker-up`/`docker-down` mentioned above:

- **Build Images**: `pdm run docker-build` - Build images after making changes
- **View Logs**: `pdm run docker-logs` - Stream logs from all containers
- **Restart All**: `pdm run docker-restart` - Quickly rebuild and relaunch all containers

Kafka Topic Management

- **Setup Topics**: `pdm run kafka-setup-topics` - Create/reset Kafka topics manually

This runs the `scripts/kafka_topic_bootstrap.py` script to idempotently create all expected topics. Note that this is automatically done on `docker-up` via the compose file, so manual use is only needed in special cases (e.g., running Kafka outside Docker).

Important: Developers are expected to use these standardized commands to ensure consistency. Using the formatter and linter ensures code meets the project's style requirements before committing.

Development Standards and Practices

Development of HuleEdu adheres to strict standards to maintain code quality, readability, and architectural consistency. These standards are documented in the internal rules (see the `.windsurf/rules/` directory) and enforced via tooling and code review:

Coding Style and Format

- **Style Guidelines**: The codebase follows PEP 8 style guidelines, automatically enforced by Ruff
- **Lint Requirements**: All code must pass lint checks (no unused imports, consistent naming, etc.)
- **Auto-formatting**: Formatting issues should be fixed by the `format-all` command
- **File Size Limits**: Maximum of 400 lines of code per file is recommended to keep modules focused (checked by the linter)

- **Organization**: Descriptive naming and clear module organization are expected

Static Typing

- **Full Annotation**: All functions, methods, and modules are fully type-annotated
- **Type Checking**: MyPy is used to ensure type correctness across the entire project
- **Maintenance**: Developers must update type hints as code evolves and resolve any MyPy warnings
- **Benefits**: Static typing catches many errors at build time and serves as up-to-date documentation for function interfaces

Testing and CI

- **Test Requirements**: Every new feature or bug fix must include appropriate tests
- **Coverage Goals**: The project maintains high test coverage including:
  - Unit tests for logic
  - Integration tests for service interactions
  - Contract tests for checking that producers and consumers of events agree on schemas

Tests are run in continuous integration, and builds will fail if tests do not pass or if coverage regresses significantly. Developers are expected to run `pdm run test-all` locally before pushing changes. Additionally, integration tests spin up dependent services (often using Docker or in-memory fakes) to verify end-to-end behavior of critical flows.

Documentation Updates

Keeping documentation in sync with the code is treated as part of the development process. When a change is made to a service or a core library, any relevant README, architectural document, or rule file must be updated in the same commit.

Example: If a new event type is introduced, its definition in `common_core` should be documented and any relevant service README should mention how the service uses that event.

The project's **Rule 090: Documentation Standards** requires that documentation be maintained as an integral part of development, and even minor changes should be reflected in docstrings or markdown docs as appropriate.

Architectural Consistency

The project has defined patterns that each service must follow, detailed in the rules under `.windsurf/rules/`. Key requirements include:

- **HTTP Services**: Must use a blueprint structure (no route definitions directly in `app.py`)
- **Background Tasks**: All long-running tasks must be idempotent to support retries
- **Cross-Service Communication**: Must go through designated integration points (Kafka or internal APIs)

During code reviews, maintainers check for compliance with these patterns. Significant deviations are not allowed unless a new pattern is being proposed and documented. This ensures that the system remains homogeneous in its architecture and that developers can navigate code across services easily.

Git Workflow

While no formal contribution guide is included here, the following practices are expected:

- **Pull Requests**: All contributions should go through pull requests with reviews
- **Commit Quality**: Commits should be granular and messages descriptive
- **Pre-commit Hooks**: The repository includes formatting/linting as a pre-commit hook (installed via `setup_huledu_environment.sh`) to catch issues early
- **Merge Requirements**: Merge decisions factor in passing CI checks (tests, lint, type check) and adherence to the above standards

Current Status of Implementation

As of now, all core microservices of the HuleEdu platform have been implemented and integrated into a functioning whole. The system is capable of ingesting a batch of essays and executing a full analysis pipeline on them.

Major Achievements

Dynamic Pipeline Orchestration

The multi-phase processing pipeline (file upload → content storage → spell check → comparative judgment → aggregation) is fully operational. The Batch Orchestrator and Batch Conductor services work in tandem to support dynamic sequencing of phases.

Observability

The observability stack is implemented uniformly across services:

- **Metrics**: Endpoints (at `/metrics`) export service-specific metrics such as request rates, event processing times, and queue lengths, which can be collected by Prometheus.
- **Health Checks**: Endpoints (`/healthz`) are in place for orchestration tools or load balancers to detect unhealthy instances.
- **Structured Logging**: Logging is structured (JSON logs or key-value pairs) and includes important identifiers (e.g., `batch_id`, `essay_id`, `correlation_id`) to make it possible to trace a single essay’s journey through the microservices by searching aggregated logs.

This readiness in observability is important for both debugging during development and for future production monitoring.

Deployment and Containerization

All services run correctly in Docker containers, and the Docker Compose setup has been tested to ensure that a new developer or tester can bring up the entire system with minimal effort. The compose file handles:

- Inter-service networking
- Environment variable wiring
- Initialization tasks like topic creation

This means the system is effectively deployable on any Docker-compatible infrastructure. While not yet deployed to a cloud environment, the necessary pieces (Docker images, network configurations, volume declarations for data) are in place, reducing the effort to move to staging or production servers. In summary, the HuleEdu platform's current implementation represents a working "walking skeleton" of the intended final system: all primary services are functional and integrated in the core workflow of processing essays. The focus so far has been on back-end architecture correctness and robustness. The system can handle the end-to-end scenario of a teacher uploading a batch of essays and receiving

analytical results. The foundation is laid for scaling up (both in terms of load and in terms of adding features).

Future Development Roadmap

While the core backend is in place, several additional services and features are planned to complete the platform's capabilities and improve the user experience. These future developments include:

AI Feedback Service (Planned)

A microservice that will generate individualized feedback for student essays using AI (LLM-based). This service would take an essay (after spellchecking, perhaps) and produce formative feedback (comments on grammar, structure, content relevance, etc.). It will likely use the LLM Provider Service to call an AI model with a prompt to generate feedback, and then emit an event with the feedback results. This service will add an "AI feedback" phase to the processing pipeline, complementing the comparative judgment score with qualitative feedback for the student.

NLP Metrics Service (Planned)

A microservice focused on computing various Natural Language Processing metrics on each essay. This could include readability scores, vocabulary complexity, sentence length variance, plagiarism detection, and other analytics. By having a dedicated service, these computationally intensive analyses can be done in parallel with other phases if appropriate. The output (various metrics and flags per essay) would be consumed by the Result Aggregator and made available in the final report.

User Management Service (Planned)

Currently, user and authentication concerns (beyond class/student relationships) are not central. A future User Service will handle platform users (teachers, students, admins), authentication credentials, and roles/permissions. This service would integrate with the API Gateway to provide JWT authentication and would manage login sessions, password resets, etc. This is crucial for a production deployment where multiple schools or institutions might use the platform with separate accounts and data isolation.

Svelte Frontend Application (In Development)

A modern web frontend (likely built with Svelte) is planned to allow educators and students to interact with HuleEdu. Through the frontend, teachers could upload batches of essays, track processing progress in real time, and review results (scores, feedback) once ready. Students might use it to submit assignments and view feedback. The frontend will communicate exclusively with the API Gateway service. Development of the UI will focus on providing a clean user experience and real-time updates via WebSockets (for example, to show a teacher a live status of their batch processing: "spellchecking 10/30 essays completed…").

WebSocket Live Updates (Planned)

Alongside the Svelte app, the full utilization of WebSockets through the API Gateway is a near-term goal. This involves finalizing a publish/subscribe mechanism (likely using Redis or Kafka streams) so that when internal events occur (e.g. an essay's analysis completes or a batch finishes a phase), a message is pushed to any connected frontends. This real-time capability will set HuleEdu apart from batch systems that only offer polling, making the experience more interactive.

Online CJ Assessment Calibration System (Planned)

As the platform relies on AI for scoring (CJ Assessment Service), we are implementing an integrated online check-and-balance system that operates within the CJ Assessment Pipeline itself. This system employs machine learning algorithms similar to traditional Automated Essay Scoring (AES) random forest models to continuously validate and calibrate AI-generated judgments in real-time. The calibration system is directly linked to the NLP

Service, which continuously analyzes essays and is intermittently retrained on the growing corpus of submitted student essays using established AES text metrics and essay quality markers.

Unlike traditional offline analytics approaches, this online system provides immediate feedback during the assessment process, flagging potential anomalies (such as essays that appear inconsistent with their assigned rankings) and generating adjustment factors that can be applied in real-time. The system produces calibration reports and reliability metrics that can be fed back into the scoring algorithm or presented to instructors for review, ensuring continuous improvement of assessment accuracy and maintaining trust in the AI-generated results.

Additional Pipeline Phases – Beyond Spellcheck, CJ, and AI feedback, other analysis phases can be incorporated into the pipeline. For example, an "NLP Enrichment" phase might annotate

essays with part-of-speech tags or entity recognition (if needed for educational feedback), or a "Peer Comparison" phase might compare a student's essay to a repository of past essays to give relative feedback. The architecture is built to accommodate new phases relatively easily by adding new services and defining their events/contracts, so the roadmap is open-ended about integrating more AI/NLP capabilities as the product evolves. Scaling and Performance – As usage grows, certain components may need to be scaled out or refactored. Future work will include performance optimizations such as: using Kafka consumer groups to allow multiple instances of worker services (Spellchecker, CJ, etc.) to share load; scaling the API Gateway and other HTTP services horizontally behind a load balancer; optimizing database interactions (adding indexes, caching frequently accessed data in RAS or CMS); and possibly partitioning Kafka topics by class or school if needed to handle very large volumes. While this is not a single feature, it is a continuous effort that will accompany the addition of users to the platform. Cloud Deployment and CI/CD – To prepare for real-world use, the project plans to containerize and deploy on cloud infrastructure (e.g. Kubernetes or a container orchestration service). CI/CD pipelines will be set up to run tests, build images, and deploy to staging/production automatically upon merges. Infrastructure-as-code (Terraform or similar) might be introduced to manage cloud resources (databases, message broker, etc.). Though largely an operations task, this is on the roadmap to transition the project from a purely local-dev setup to a live service. This roadmap is subject to refinement as the project progresses and user feedback is gathered. However, it gives a clear direction: the immediate next steps focus on front-end integration (API Gateway and Svelte UI), richer analysis features (AI feedback, NLP metrics), and robust user management. These additions, combined with the strong backend foundation already in place, will move HuleEdu toward a production-ready state suitable for pilot programs and eventually larger deployments.

Documentation and Further Information

This README provides a high-level overview of the HuleEdu platform. For more detailed information, consult the following resources:

Service-Specific Documentation

- Each microservice has its own README file in its root directory (e.g., `/services/content_service/README.md`).
- These files contain detailed information about the service's responsibilities, API endpoints (if applicable), configuration options, and specific deployment notes.
- If you are working on or using a particular service, refer to its README for in-depth information.

Architecture & Design Documents

- The `documentation/` directory contains high-level design documents, product requirement documents (PRDs), and technical plans for various features.
- Notable files include detailed discussions of the state machine design, the reasoning behind certain architectural choices, and plans for future phases.
- These documents are useful for understanding the rationale behind the implementation and for onboarding new contributors to the system's design philosophy.

Architectural Decision Records (ADRs)

- Key architectural decisions are documented in ADRs, located in the `/docs/adr/` directory.
- These records explain the context, decision, and consequences for significant choices, such as the adoption of event-driven architecture, the selection of RabbitMQ, or the design of the processing pipelines.

Development Rules

- The `.windsurf/rules/` directory defines the project’s development rules and standards in a structured format.
- This covers everything from project structure conventions to coding style, error handling patterns, and documentation requirements.
- The `000-rule-index.mdc` file provides an index of all available rule files.
- Developers should familiarize themselves with these rules, as they codify best practices and are treated as part of the code review criteria.
- They also ensure that as the team grows, the codebase remains uniform and maintainable.

Changelog

- The `CHANGELOG.md` or commit history can be consulted for a chronological record of major changes and feature additions.
- This can help in understanding how the system evolved to its current state.