**CI/CD Process and Project Documentation**

**Project Overview**

This project implements a backend system that simulates a shared spreadsheet-like application, powered by FastAPI, DuckDB, and Redis. The CI/CD pipeline ensures code quality, automates testing, and validates data integrity for a production-ready deployment.

This document is structured to provide clarity by covering coding implementation details, tools used, and a structured ticketing system.

**1. Tools, Packages, and Libraries**

**Backend Development:**

FastAPI: Python-based web framework for creating APIs. Includes built-in support for **Pydantic** models for data validation.

Uvicorn: ASGI server for running the FastAPI application. Light-weight and Production ready.

Visual Studio Code: Development Environment for coding, debugging and testing

**Database and Data Management:**

DuckDB: In-memory analytics database for querying. Used as backend for managing data/views.

Redis: Handles request and response streams.

Pydantic: Ensures data validation and integrity in API endpoints.

Faker: Generates synthetic data for testing and populating DuckDB tables.

**Dependency Management:**

Poetry: Handles dependency installation and versioning. pyproject.toml ensures reproducible buils. Preferred than pip for better dependency resolution, virtual environment isolation.

**Code Quality and Formatting**

Pydantic**:** Validates request and response data. Ensures clean API data processing

Flake8: Linter to enforce PEP 8 standards.

Black: Code formatter for consistent style.

Ruff: A faster linter and formatter, combining functionalities of Flake8 and Black.

**Testing:**

Pytest: Executes unit tests for database operations, Redis streams and API endpoints. Supports integration testing for end-to-end validation.

Coverage.py: Measures test coverage. Ensure all code paths are settled.

**CI/CD Automation:**

GitHub Actions: Automates linting, testing, and deployment pipelines. Runs workflows for every push and pull\_request.

**Deployment Tools:**

Uvicorn: Deploys the application in production. Supports multi-threading for concurrent user requests.

**Additional Tools:**

Postman: Used for testing API endpoints manually. Simulates user requests and validates server responses.

Pre-commit Hooks: Automates code quality checks (linting, formatting) before committing code.

**2.Coding Implementation:**

**File 1: data.py (Data Creation)**

Purpose: Create and populate DuckDB tables (product, customer, and sales).

Highlights:

- Uses `Faker` to generate synthetic data for tables.

- Creates `sales\_summary\_by\_product\_family` table with grouping sets.

- Implements pivot and unpivot views for data transformation.

**File 2: DuckDBManager.py**

Purpose: Manage DuckDB operations.

Highlights:

- Implements a singleton class to handle a single read-write connection.

- Methods for proportional rebalance, dependency updates, and recalculating summaries.

**File 3: logger.py**

Purpose: Centralized logging for API, Redis, and database operations.

Highlights:

- Logs API requests, Redis interactions, and DuckDB queries.

- Ensures logs are written to separate files for better tracking.

**File 4: mainapi.py**

Purpose: Contains FastAPI routes for handling cell updates.

Highlights:

- Processes `/update\_cell` API requests.

- Validates input using Pydantic models.

- Updates DuckDB tables and handles Redis request/response streams.

**File 5: redislistener.py**

Purpose: Listens to Redis streams for processing requests and broadcasting updates.

Highlights:

- Reads from ‘request\_duck’ stream and writes responses to ‘response\_duck’.

**3. CI/CD Pipeline Workflow**

Step 1: Dependency Management

1. Use `Poetry` to manage dependencies.

- Install dependencies: poetry install.

- dependencies managed via pyproject.toml

- Ensures consistent env across. Simplifies version upgrades and locking.

Step 2: Linting and Formatting

1. Use `ruff` for faster linting: `ruff check app`. It is significantly faster than Flake8 and Black, ideal for large codebases. It performs both linting(Flake8) and formatting(Black). Flake8 demonstates PEP 8 compliance and error detection.
2. Intending to use Flake8 + Ruff: Ensure deep static analysis and modern, fast linting and formatting.

Step 3: Testing

1. Write test cases using pytest.

2. Run tests using pytest and measure coverage with pytest-cov.

Step 4: GitHub Actions Configuration

1. Automate CI/CD pipeline:

Workflow triggers on every push and pull request

* Checkout Code, Install dependencies via poetry, link codebase with Ruff and Flake8, Run tests using Pytest.