AC215 Milestone 4 Deliverables

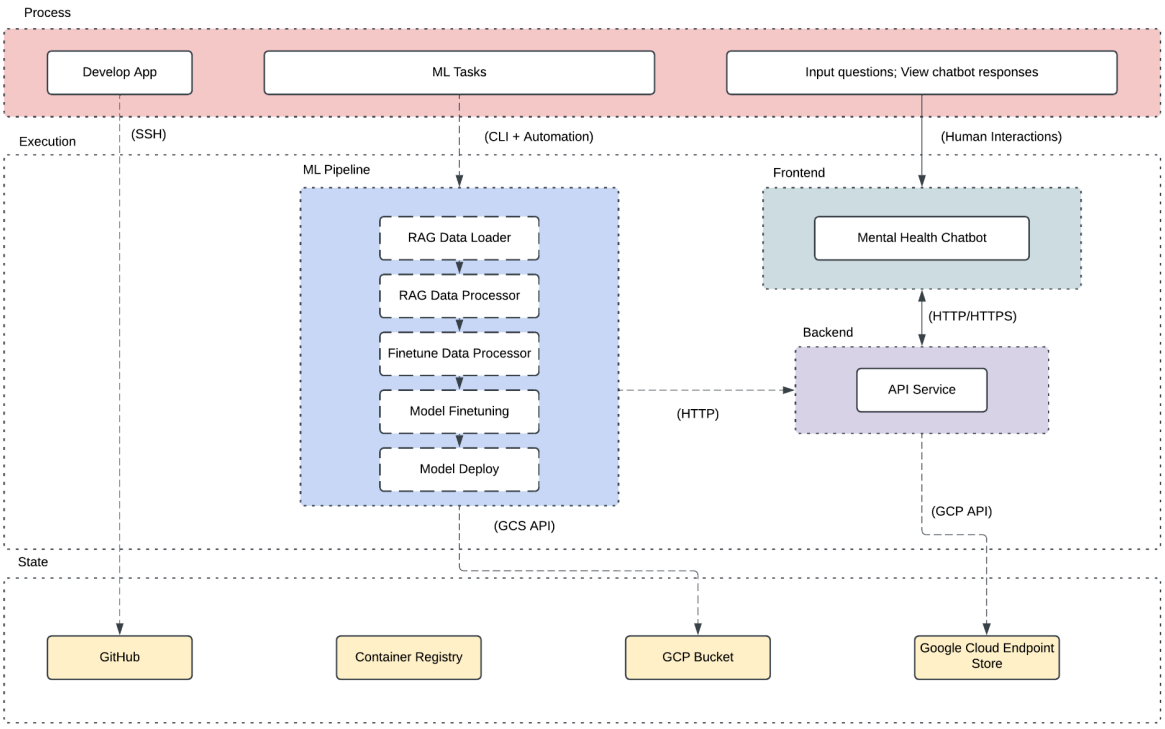
Nina Mao, Yunxiao Tang, Jiayi Xu, Xinjie Yi

Overview

This document explains the application design, including its architecture, user interface, and code organization. It covers both the high-level solution architecture and the specific technical architecture to ensure a comprehensive understanding of the system components and their interactions.

1. Solution Architecture

The solution is designed to deliver a mental health chatbot application that integrates RAG, finetuning and data pipelines to generate contextually accurate responses. The architecture is divided into three main layers:



Process Layer

- Develop App: Development and deployment of the frontend and backend components.
- ML Tasks: Includes data preprocessing, model fine-tuning, embedding generation, and deploying the chatbot.
- User Interaction: Users can input mental health-related queries via a web interface to receive real-time chatbot responses.

Execution Layer

- ML Pipeline: Automates the processing of data and ML tasks. It includes:
    - RAG Data Loader: Fetches raw academic papers for data ingestion.

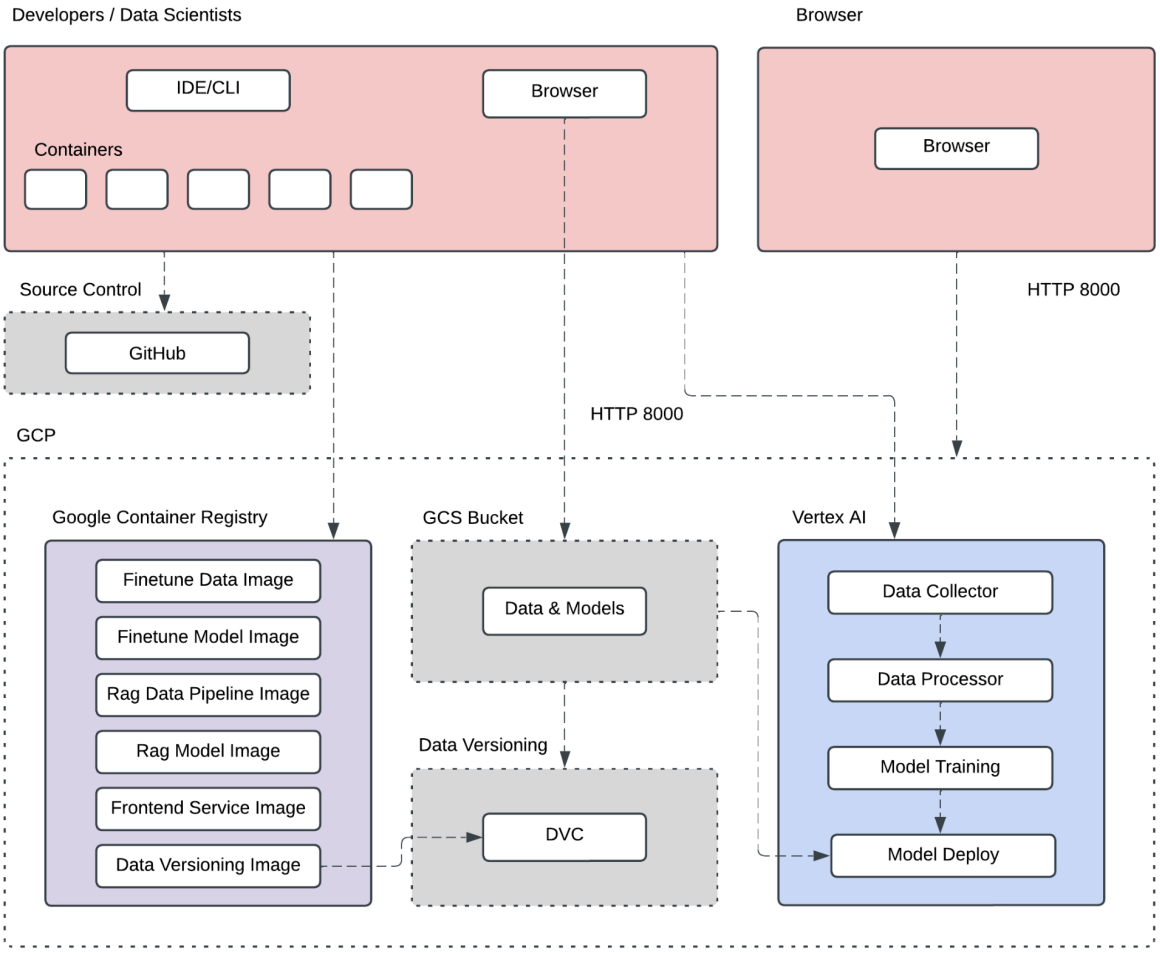
    - RAG Data Processor: Prepares the text data for embedding generation and storage in a vector database.
    - Fine-Tune Data Processor: Handles the preprocessing of mental health-related Q&A conversations for model fine-tuning.
    - Model Fine-Tuning: Trains the base model on the preprocessed dataset.
    - Model Deployment: Deploys the fine-tuned model to a cloud endpoint for real-time inference.
- Frontend and Backend:
    - Frontend: A user-facing mental health chatbot interface for interacting with the system.
    - Backend: Includes API services for processing user inputs and forwarding requests to the ML model.

State Management Layer

- GitHub: Manages the source code and versioning.
- Google Container Registry: Stores Docker container images for each system component.
- Google Cloud Storage (GCS): Hosts preprocessed datasets, embeddings, and models.
- Google Cloud Endpoint Store: Manages deployed model endpoints for real-time inference.

2. Technical Architecture

The technical architecture focuses on the implementation details and system-level design of the chatbot application. It outlines the integration of various components, such as data pipelines, model fine-tuning, and deployment workflows, to ensure functionality and scalability.



Technologies and Frameworks

Backend:

- Python-based Flask API for serving requests.
- LangChain for RAG (Retrieval-Augmented Generation) workflows.
- Google Cloud AI Platform for model hosting.

Frontend:

- HTML templates with Flask integration.
- Interactive chatbot interface with real-time input/output.

Data Pipelines:

- RAG Workflow: Chunking, embedding generation, and storing embeddings in ChromaDB.

- Fine-Tuning Pipeline: Data preprocessing with pandas and scikit-learn, followed by model fine-tuning using Hugging Face Transformers.
- Data Versioning: DVC (Data Version Control) for managing dataset versions.

Containers and Orchestration:

- Docker for containerization of all services.
- Docker Compose for orchestrating multiple containers.

Cloud Services:

- Google Cloud Storage (GCS) for data storage.
- Google Cloud Container Registry for Docker image storage.
- Vertex AI for model deployment and monitoring.

Code Organization

See detailed code organization in README.md

Design Patterns

- Microservices Architecture: Each major task (data preparation, model fine-tuning, RAG pipeline, etc.) is containerized and modularized.
- Pipeline-Oriented Workflow: Data flows through a series of clearly defined stages in the ML pipeline, ensuring traceability and reproducibility.
- Cloud-Native Deployment: All critical assets (e.g., models, embeddings, datasets) are stored and deployed using Google Cloud services.

3. Interaction Flow

Frontend Interaction

- Users interact with the chatbot via the browser-based UI, typing queries and receiving responses in real-time.

Backend Workflow

- The chatbot UI sends the user's query to the backend API.
- The backend processes the query and forwards it to the RAG workflow or the fine-tuned model.
- The model generates a response and sends it back to the frontend.

ML Pipeline Automation

- Data preprocessing, fine-tuning, and embedding generation are automated using CLI commands integrated into Docker containers.

3. Test Documentation

Testing Tools

- PyTest: Utilized for testing integration and system functionalities.
- Unittest: Used for unit testing individual modules and their interactions.
- Mock: Used extensively for simulating external dependencies, such as VertexAI and Chromadb to isolate test environments.

Testing Strategy

1. Unit Tests

Unit tests validate individual components in isolation:

TestFineTuningScript

- test_train: Ensures the fine-tuning process is executed with correct parameters and produces expected outputs.

TestLLMFineTuningData

- test_prepare: Verifies finetunedata preparation splits intents into training and testing datasets and writes them to correct files.

TestPreprocessRag

- test_generate_query_embedding: Tests query embedding generation and ensures embeddings match expected structure.
- test_generate_text_embeddings: Validates text embedding generation for multiple input chunks.
- test_load: Ensures embeddings, chunks, and metadata are correctly added to the ChromaDB collection.
- test_query: Tests querying embeddings and validates results returned from the ChromaDB collection.
- test_get: Verifies document retrieval from ChromaDB using filters.

TestDownloadFunction

- test_download: Ensures data for the RAG system is correctly downloaded from Google Cloud Storage and handled appropriately.

2. Integration Tests

Integration tests ensure that multiple components work together as expected:

- test_integration: Validates interactions with the ChromaDB client in the RAG process by testing the following:
    - Text embeddings are correctly loaded into the collection with proper metadata and IDs.
    - Queries return expected results from the collection.
    - Document retrieval works with specified filters and returns the expected data.

3. System Tests

System tests Covering user flows and interactions:

- test_chat_route: Pretends to be a user and tests the /chat endpoint for a valid input message and verifies the correct response is generated.
- test_no_input_route: Pretends to be a user and tests the /chat endpoint for a missing input message and ensures an appropriate error response is returned.

Test Coverage Report

Below is the code coverage summary from the most recent test suite run:

```
---------- coverage: platform darwin, python 3.12.0-final-0 ----------
Name                                  Stmts   Miss  Cover
---------------------------------------------------------
finetune_data/prepare_data.py            66     23    65%
finetune_model/finetune.py               50     21    58%
rag_data_pipeline/dataloader.py          26      8    69%
rag_data_pipeline/preprocess_rag.py     185    101    45%
service/app.py                           58     24    59%
tests/conftest.py                        12      0   100%
tests/test_integrate.py                  37      0   100%
tests/test_system.py                     25      0   100%
tests/test_unit.py                      102      0   100%
---------------------------------------------------------
TOTAL                                   561    177    68%
```

Instructions to Run Tests Manually

- Navigate to the src directory: cd src
- Export the environment variables: source env.dev
- Install the conda environment: conda env create -f environment.yaml
- Generate the test report: pytest --cov=. tests/