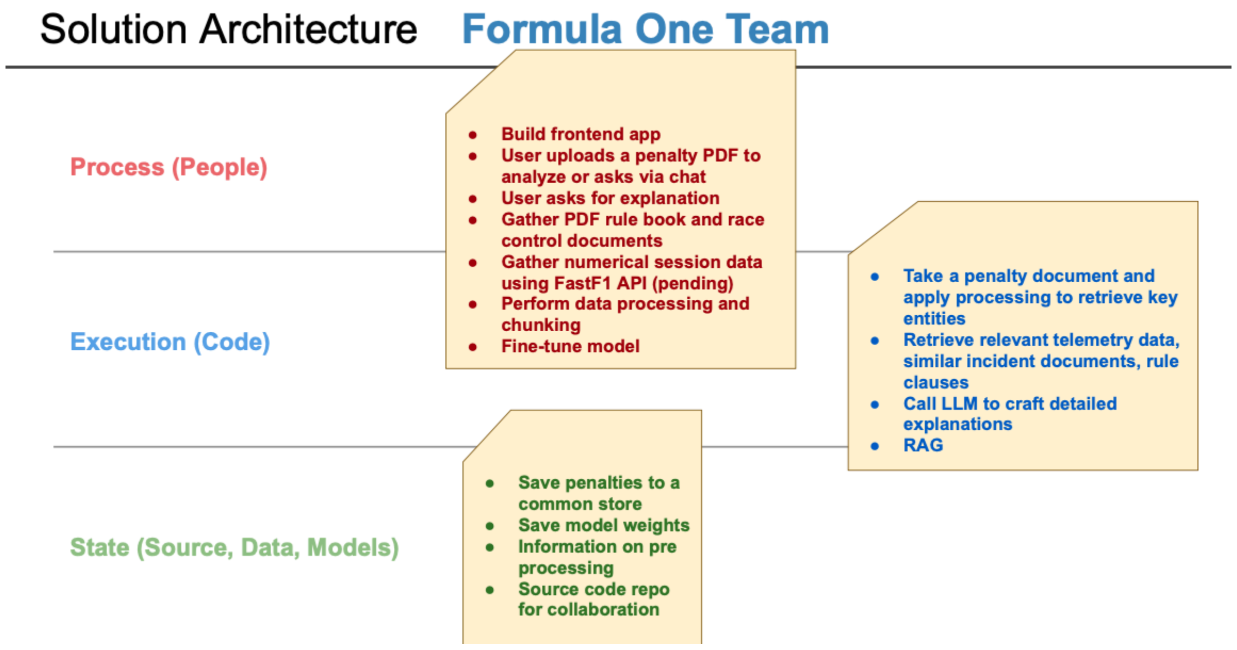
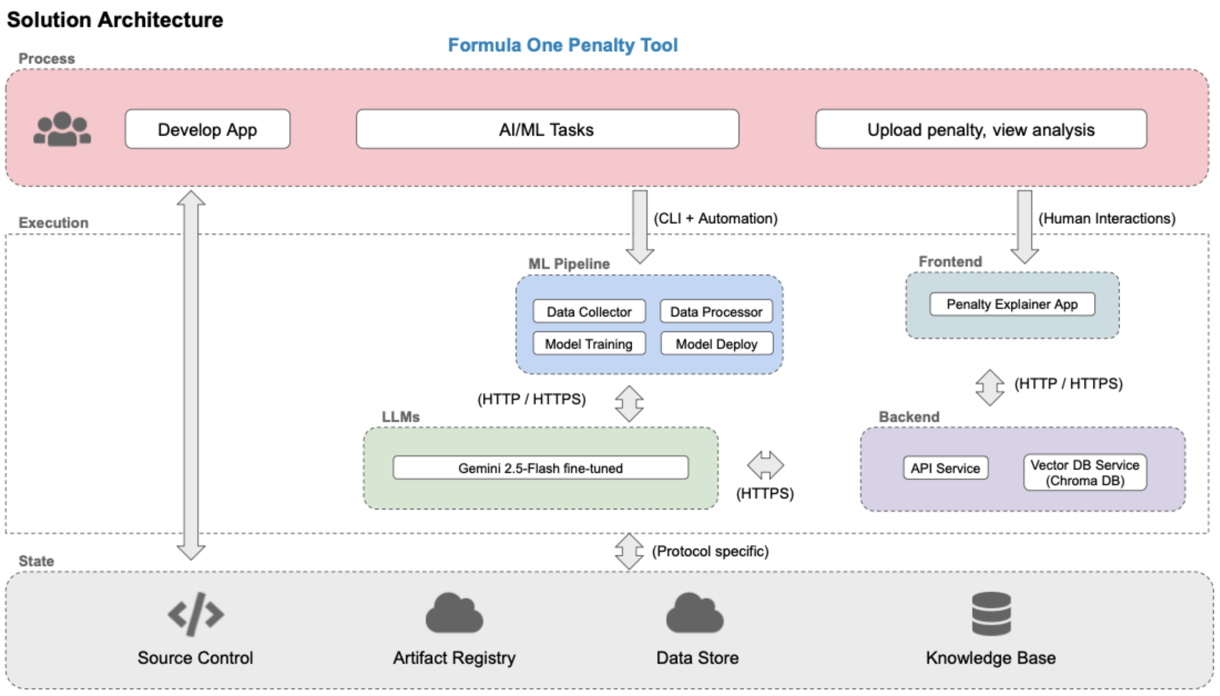
Application Design Document
Formula One Penalty Tool

Solution Architecture

This architecture is structured around three layers: Process (People), Execution (Code), and State (Source, Data, Models).


Solution Architecture
Formula One Team
Process (People)
Execution (Code)
State (Source, Data, Models)
• Build frontend app
• User uploads a penalty PDF to analyze or asks via chat
• User asks for explanation
• Gather PDF rule book and race control documents
• Gather numerical session data using FastF1 API (pending)
• Perform data processing and chunking
• Fine-tune model
• Take a penalty document and apply processing to retrieve key entities
• Retrieve relevant telemetry data, similar incident documents, rule clauses
• Call LLM to craft detailed explanations
• RAG
• Save penalties to a common store
• Save model weights
• Information on pre processing
• Source code repo for collaboration



Process (People)

The layer that captures human-driven activities involved in building and using the system.

- Develop a frontend application that enables users to upload penalty PDFs or ask questions through a chat interface
- Collect data, including FIA rulebook PDFs and steward decision PDFs
- Preprocess and chunk documents
- Fine-tune the LLM on example penalties, rule citations, and historical race incidents
- User interaction, where users upload penalties, request explanations, and view generated analyses.

Execution (Code)

The layer that corresponds to the logic that runs the penalty tool.

- ML Pipeline: Document processing extracts information from a penalty document
- LLM: Using a fine-tuned Gemini 2.5 Flash model to craft rule-grounded explanations
- Frontend: The frontend application exposes a UI for human queries and to display system outputs/responses
- Backend: Expose APIs for the document analysis and for the orchestration of LLM calls

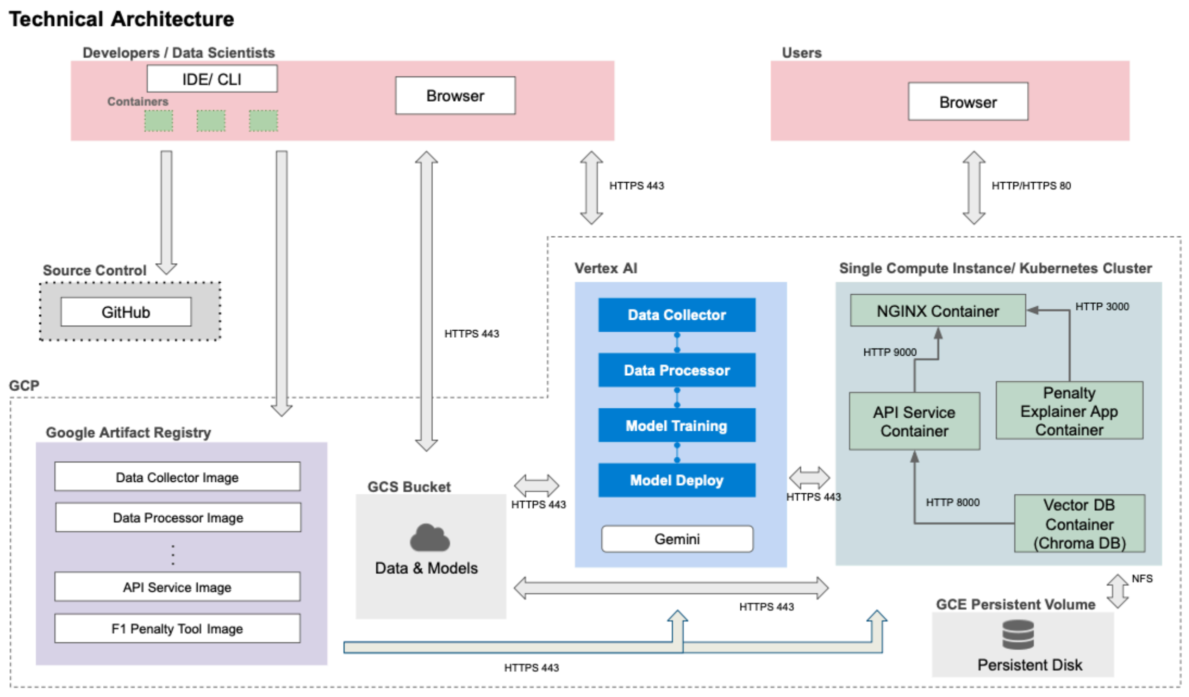
State (Source, Data, Models)

The layer that defines the persistent resources that the system depends on.

- Source control: Source code repository for all application components.
- Knowledge base: Stored in a GCS bucket, includes rulebooks, historical penalties, and vector embeddings.
- Artifact registry: To store container images including the containers needed for the data and RAG pipelines, as well as the APIs.
- Data store: For any GCP persistent disks that may be needed once the project is deployed to run on GCP resources and not locally.

Technical Architecture

This diagram describes the concrete technologies and components that implement the solution. It shows how developers work, how the AI system is built and deployed, and how the application runs in production.



Development Workflow

- Developers use an IDE and CLI to develop and commit code to GitHub (used for source control)
- Each service is containerized and pushed to Google Artifact Registry.

ML Pipeline (Vertex AI)

- Data collector gathers rulebook PDFs and steward documents corresponding to penalties.
- Data processor cleans, chunks, and embeds documents.
- The model training lives on Vertex AI and fine-tunes Gemini 2.5-Flash.
- Model deployment publishes the tuned model for inference.

Deployment Infrastructure

Most of this work will be completed as part of Milestone 5

- The application will run on a Kubernetes cluster with multiple containers:
  - NGINX container: To serve the UI and route traffic
  - Penalty explainer app container: The web frontend
  - API service container: Handles retrieval and LLM orchestration
  - Vector DB container: Holds embeddings for rulebooks and prior penalties
- A GCE Persistent Disk will store information that's needed to preserve the state of the container.

Networking

- There’s different ports for each connection:
    - Frontend <-> Backend: HTTP/HTTPS (port 3000/9000)
    - Backend <-> VectorDB: HTTP (8000)
    - Backend <-> Vertex AI/LLM: HTTPS 443
    - Developer access to artifact registry and GCS: HTTPS 443

Code Organization

/

- Dockerfile
- other Docker-related files including docker-entrypoint, docker-compose, docker-shell, pyproject

/src/datapipeline

- web scraping for penalty documents
- PDF processing

/src/finetune

- data processing for training JSONL files

/src/rag

- rag pipeline including chunking, embedding, storing, and query
- backend APIs

/reports

- deliverables for past milestones