Technical Design & Notes

1.System Architecture

The solution is implemented using a modular, service-oriented architecture optimized for clarity, testability, and incremental extension. Every responsibility is isolated into its own layer to enable clean separation of concerns.

1- API Layer (Fast API)

Located in: app/routers/

- Exposes the main endpoints:
    - /classify → to test the intent classifier pipeline standalone (not used by the UI)
    - /generate → full assistant pipeline (classification → retrieval → LLM reasoning)
    - /feedback → user evaluation / thumbs up or down
    - /logs → access filtered JSONL logs
- They call **orchestrator functions** (services) that perform the core logic.
- Ensures consistent request/response schemes using Pydantic models.

2- Configuration Layer

Located in: app/settings.py

- Centralized configuration for:
    - Model paths
    - Knowledge base directories
    - LLM model selection
    - Thresholds (RAG chunk limits, OOS confidence threshold)
    - Logging configuration
- Uses deterministic path resolution to avoid uvicorn reloader inconsistencies.
- Eliminates reliance on magic strings in code.

3- ML Layer (Intent Classification)

Located in: app/ml/ and app/utils/intent_classifier.py

A hybrid classification pipeline integrating:

- TF-IDF vectorizer
- SentenceTransformer embeddings for semantic fallback or augmentation.
- Logistic Regression classifier
- Calibrated probabilities (via CalibratedClassifierCV)
- Out-of-scope detection using: if max_proba < OOS_THRESHOLD: return "out_of_scope"
- Model saved under app/ml/models/intent_pipeline.joblib

4- Static Knowledge Base Layer (RAG)

Located in: app/data/chroma/ and app/data/kb_md_files/

- Markdown documentation under kb_md_files/ contains:
    - onboarding guides,
    - escalation policy,
    - deployment procedures,
    - team structure,
    - dev environment setup,
    - code review rules.
- Chunking pipeline (chromaKB_build.py):
    - Splits markdown into overlapping windows.
    - Attaches metadata: intent, source_file, chunk_index.
    - Encodes chunks via SentenceTransformer.
    - Stores vectors into a persistent ChromaDB instance.

5- **Retriever**

Located in: app/utils/retriever.py

- Handles:
    - similarity search,
    - intent-based filtering,
    - retrieval scoring,
    - top-k selection,
    - fallback logic.
- Ensures the LLM receives grounded and minimal context.

6- Dynamic Data Layer (Mock APIs)

Located in: app/mock_api.py and JSON files under app/data/

- Simulates APIs for:
    - employees
    - deployments
    - Jira tickets
- API-like interfaces return structured JSON objects usable by the LLM.
- All dynamic intents call this layer.

7- LLM Reasoning Layer (Gemini)

Located in: app/llm/

- Responsible for:
    - formatting system + user prompts,
    - injecting KB chunks or dynamic data,
    - enforcing response structure (JSON / markdown / bullet points),
    - safety constraints,

    - grounding rules (“only use provided context”).
- Abstracted as a thin client so switching models (OpenAI → Gemini → Llama 3) is a single-file change.

8- Pipeline Orchestrator Layer

Implemented inside generate router & service helpers.

Steps:

1. **Intent classification**
2. **Routing**
    - static (KB)
    - dynamic (mock API)
    - out-of-scope
3. **Retrieval**
    - intent-filtered chunk search
    - semantic top-k ranking
4. **LLM selection** & prompt build
5. **LLM inference**
6. **Trace logging**
7. **Response packaging**

9- Observability & Logging Layer

Located in: app/utils/logger.py and app/logs/

system implements a robust logging pipeline using structured **JSONL** log files.

Each interaction logs:

- timestamp
- request ID
- endpoint

- user query
- predicted intent + confidence
- retrieval results (chunk IDs, similarity scores)
- LLM model used
- response time
- error trace (if any)
- final assistant output

JSONL was chosen because:

- append-only
- easy to query with grep/pandas
- compatible with log pipelines (ELK, Loki, Datadog)
- simple enough for local debugging

2. Key Design Decisions

2.1 Using a Hybrid Intent Classifier

Instead of relying solely on TF-IDF or embeddings, a hybrid approach was chosen because:

- **TF-IDF** captures sharp lexical cues (e.g., "JIRA", "deployment", "review").
- **Embeddings** capture semantic similarity and paraphrasing.
- **Combining both** improves robustness across formal, informal, and ambiguous queries.

The hybrid feature vectors are balanced and fed to a calibrated Logistic Regression model to yield interpretable probabilities.

2.2 Multi-Intent Routing

Real developer queries often implicitly require more than one data source.
Therefore, instead of selecting only the top intent, the system:

- Returns all intents above a probability threshold.
- Applies special rules for **out_of_scope** to prevent false positives.

- Allows the endpoint to route to:
    - **Static KB retrieval**
    - **Dynamic lookup**
    - **Both**, depending on which intents are selected.

This increases accuracy for multi-facet questions such as *"Who deployed payments last week?"*.

2.3 Static KB Chunking Strategy

Markdown files are chunked using:

- Headings as primary separators
- Recursive splitting for sections exceeding ~500 tokens
- Metadata embedding (intent/document name, section title)

This ensures:

- High semantic coherence in chunks
- Better ranking during retrieval
- Traceable citations for the LLM's final answer

2.4 Dynamic Data Retrieval Without Over-Engineering

Given the constraints, dynamic data is simulated via:

- **Simple structured functions** for employees, Jira tickets, and deployments
- Querying logic based on extracted parameters (name, ticket ID, service name)

This provides realistic behavior without external dependencies.

3. ML Orchestrator & Pipeline Flow

The orchestrator logic inside the /generate endpoint coordinates the entire process:

1. **Receive user query**
2. **Intent classification**
    - Multi-intent

    - Calibrated probabilities
    - Out-of-scope dominance logic applied
3. **Determine required data sources**
   Based on mapping from intents → resource type
4. **Context retrieval**
    - From ChromaDB for static docs
    - From mock dynamic API for JSON sources
5. **Create LLM prompt**
    - System message
    - Retrieved context (tagged with metadata)
    - User query
6. **LLM generates structured response**
7. **Response validated with Pydantic**
8. **Everything logged in JSONL**

4. Intent Training & Classification Strategy

4.1 Dataset Generation

Because the dataset had to simulate real Harri developer questions, examples were generated per document and dynamic source. Challenges included:

- Mislabeling in synthetic questions
- Weak out-of-scope representation
- Overlapping semantics (employees vs team structure)

4.2 Training Strategy

- Train/validation split with stratification
- Hybrid feature combination (TF-IDF sparse + embedding dense)
- Logistic Regression classifier

- Probability calibration using CalibratedClassifierCV
  to improve confidence reliability
- Evaluation across each class to check for imbalance issues

4.3 Multi-Intent Inference Rules

- All intents ≥ threshold included
- out_of_scope dominates if ≥ 0.5
- Intent → source mapping used to determine retrieval path

5. Observability & Logging

A lightweight but effective observability pipeline was implemented.

5.1 JSONL Logging

Each user interaction produces a structured log entry containing:

- Timestamp
- User query
- Classified intents + probabilities
- Retrieval actions taken (KB chunks, dynamic JSON queries)
- LLM response (structured)
- Any errors or fallbacks triggered

5.2 Debug Panel in Streamlit

The UI exposes:

- Last query
- Intents predicted
- Retrieved chunks
- Dynamic lookup results
- Final LLM reasoning

This gives transparency into the system's reasoning path—critical for debugging ML-driven pipelines.

6. Handling Limits

To operate reliably within the complexity of a hybrid ML+LLM system, several safeguard mechanisms were introduced:

- Fallback when no intent exceeds threshold
- out_of_scope dominance to prevent hallucinated routing
- Max chunk limit to prevent context overloading
- Validation errors handled gracefully for LLM output
- Timeout & error logging for each step
- Strict grounding rules in the system prompt to prevent hallucinations

7. Assumptions

The solution is built under the following assumptions:

- Only documents and JSON data provided in the challenge constitute valid knowledge.
- Dynamic data is mocked and static, not updated in real time.
- No authentication or user roles are required.
- Multi-user concurrency is not expected at large scale.
- LLM calls will remain deterministic enough given the strong system prompt constraints.
- Data coverage is representative of typical Harri developer workflows.