Self-Reflection - Harri Dev Team AI Assistant

One of the most challenging parts of this project was generating a good dataset for the intent classifier. I wanted the examples to realistically reflect what Harri developers might actually ask in day-to-day work, not just synthetic phrases. To do this, I used ChatGPT to generate questions for each document and data source, then aggregated these into a single dataset. In practice, this turned out to be trickier than expected: some examples ended up labeled with the wrong intent, and my out-of-scope coverage was not very representative of the kinds of "random" or non-technical questions a real user might ask. When I tested the trained model with queries from the README and other realistic prompts, some failures were directly traceable to these issues in the dataset, not just the model.

On the modeling side, I spent a lot of time iterating on the intent classifier. I first trained a TF-IDF + logistic regression model and noticed it performed poorly on some classes, especially where wording overlapped heavily or where synonyms were used (for example, team vs employee-related questions). The model was very sensitive to exact phrasing. I then tried an embedding-based approach with logistic regression, which handled semantic similarity better but sometimes blurred the distinctions between closely related intents and still produced overconfident probabilities. That led me to treat TF-IDF and embeddings as complementary: TF-IDF is good at sharp lexical signals, while sentence embeddings capture higher-level meaning. I experimented with combining the sparse TF-IDF features and dense embedding features carefully (paying attention to dimensionality and scaling) and saw improvements, especially on the intents that were previously weak. Getting this hybrid setup stable and interpretable took time.

Choosing a clean project structure was another concrete challenge. I didn't want everything to live in a single "utils" file. Instead, I tried to group related concerns together: ML code under a dedicated ml area, knowledge base building and retrieval under core, API routers separated by responsibility, and small helper modules for things like logging and mock dynamic data. Some of this restructuring was actually forced by practical issues like joblib failing to load models when classes were defined in notebooks, but it ended up improving clarity and making the pipeline easier to reason about.

A large portion of my time went into designing and refining the intent handling logic itself: deciding when something should be treated as out_of_scope, when that label should "dominate" other intents, how to interpret multi-intent probabilities, and when to include multiple intents versus just the top one. I had to make explicit assumptions and encode them in thresholds and rules (for example, how high out_of_scope needs to be before I ignore the rest). Getting this behavior to feel sensible across different test queries required a lot of trial and error.

Building the vector database for the knowledge base was comparatively smoother, mainly because the markdown documents were already well structured with clear headings. I still had to make decisions about how to chunk them splitting by headings and then further splitting sections above a certain length. Those choices influenced retrieval quality, so I iterated a bit there as well. Designing the system prompt also took time; at first, there were contradictions between what I asked the LLM to return and the Pydantic response model. This led to validation errors when the response didn't match the expected schema, and I had to refine the instructions until Gemini consistently produced well-structured, grounded answers with source references.

What I Would Improve With More Time:

If I had more time, there are several areas I would push further:

1. API security and multi-user handling:
   I would add authentication/authorization around the API and design proper multi-user session handling, so different users' state and conversations are isolated in a production-ready way (e.g., with session IDs and a backing store like Redis or a database).

2. Parameter and retrieval calibration:
   I would more systematically tune parameters such as the number of chunks to retrieve from the knowledge base and intent probability thresholds, using evaluation runs rather than manual intuition.

3. Smarter dynamic parameter extraction:
   For dynamic data, I would implement a small LLM-based component to extract structured parameters (like employee name, service name, ticket ID) from the user query and use those to filter the JSON data, instead of always returning full JSON objects. From the README, my understanding is that returning the full JSON is acceptable as a mock, but a more targeted lookup would be closer to a real system.

4. Richer logging and learning from usage:
   I would store logs in a queryable database and use them to understand where the assistant struggles. This could reveal new intents to add, gaps in the knowledge base, or common queries that are currently misclassified. On top of that, I would like to build a small model or pipeline that periodically scans the logs, detects misclassified queries, suggests corrected labels, and feeds them back into the training set for retraining.

5. Async and performance improvements:
   Since the Gemini API supports async usage, I would refactor the integration to use asynchronous calls. This would significantly improve performance and scalability when multiple users query the assistant concurrently.

6. Automated KB and vector store updates:
   Finally, I would implement an automated process (for example, using GitHub Actions or a file-change watcher) to rebuild or update the Chroma vector store when the underlying markdown files change, so the assistant always reflects the latest documentation without manual intervention.

7. Self-Reflection document itself

   With additional time, I would also expand this self-reflection into a more comprehensive retrospective that documents not only the final system but the reasoning, false starts, and trade-offs made along the way. A deeper write-up would better capture the challenges encountered, the approaches that were evaluated and discarded, and the insights gained from iterative experimentation. Many of these decisions reflect how I think about problem-solving in real projects, and I would welcome the opportunity to discuss them in detail during a follow-up conversation if the process continues.

Overall, this project forced me to think carefully about data quality, intent modeling, routing between static and dynamic sources, and how all of that shows up in a user-facing assistant. The areas I struggled with the most dataset design, hybrid features, and multi-intent handling are also the places where I learned the most about what it takes to turn a prototype into something that behaves reliably in realistic scenarios.