AI-Powered Email Assistant: An Intelligent System for Email Management

Postgraduate Major Project - MOD002726

SID 2367793

September 2025

Contents

List of Figures

List of Tables

Abstract

This project explores open-source large language models (LLMs) to summarize and analyze emails, with a focus on simplicity and accessibility. Using Ollama, this project integrates different LLMs like mistral, DeepSeek —to extract key insights from loads of mails and present summary, visual analysis, classification and prioritization. Results appeared via a Streamlit-based UI, allowing easy, real-time user interaction.

The work involved aquiring email data, preprocessing it, crafting prompts, creating functions evaluating output for clarity, brevity, and relevance. Mistral excelled at speed and concise summaries and extracting data for analysis while llama3.2 offered better and stable responses to the user queries. The system eases the handling of large volumes inbox, extending its potential applications in almost every corporate, admin, and customer service sector.

Some of key learnings gained are Problem solving, RAG, LLM integration, UI design and development, and evaluation. The project serves as a step towards how AI-based text analysis can improvise efficiency in everyday lives.

1 Introduction

1.1 Scope of the Research

This project aims to build a tool for managing emails. The objective is to minimize time spent on large chunks of mails and emphasize critical information and organize inboxes from becoming just an information garage. Emails are loaded and processed, meaning is derived, priorities are emphasized and relevant data is presented to the user.

The study has five principal objectives. Identify the current challenges in email management initially. Secondly, examine how AI, LLM, RAG and visual dashboards might modify the sorting and understanding of messages. Third, evaluate whether a chatbot can function as a real-time assistant rather than introducing more complexity. Finally, evaluate the tool's usability, accessibility, and compatibility..

1.2 Context and Background

We stand on the shoulders of giants (recent advancements). Demis Hassabis (Google DeepMind CEO) has publicly talked about a "next-generation email" assistant that can sort, reply in your own writing style, and reduce time lost to email backlog (Hassabis 2025). The manual effort is diminished when the laborious activities of sorting and filtering are managed by automated processing. AI today is more capable on context-awareness. Google has expanding features like email summarization, smart replies, and generative models to mimic tone. Data visualization converts imperceptible communication flows into readily viewable charts and indicators. A chatbot functions as the flight officer by addressing inquiries and aiding the user with routine duties. Emails are classified into user-friendly categories, including spam, work, personal, and promotional. The security perimeter comprises multi-factor authentication and encryption.

The predominant tools offered on the market mostly execute basic functions such as flagging, sorting, and filtering. They are rigid, often unwieldy, and ill-equipped to manage the volume and intricacy of modern email communications.

1.3 Need for the Project

Emails still overwhelming as people miss critical messages, small tasks diminish in gaps, and they suffer accountability. Even though Gmail and others have smart filters and replies, that's only part of the solution.

What Google and Hassabis have said reinforces this gap. He's said publicly that he'd pay lots of money to get rid of email overload. He is pushing for tools that can do more than just sort — tools that can feel like an assistant, automatically reply, catch urgency, mimic style. Inbox overload is not just irritating but also detrimental. Teams squander time pursuing threads instead of making decisions, priorities become ambiguous, and critical tasks are overlooked amidst promotions. Legacy systems merely address the issue rather than resolving it.

The inequity is evident. We necessitate a system capable of understanding the content as well as perceiving the volume. The intelligence derives from AI and NLP, the rapidity from real-time prioritization, the interactivity via chatbot integration, and the overall safety is guaranteed by stringent security measures. When amalgamated, these transcend mere accessories. Businesses inundated by email create a comprehensive ecology.

1.4 Methodology

The research consists of a combination different phrases. Literature review was initial phase, exploring existing knowledge and identifying pits. The system was designed and expanded from that point from incorporating a chatbot for real-time assistance, then integrated dashboards for visual analytics, and a prototype utilizing streamlit based interface. Then several testing and improvisations were done and analyzed its strengths and flaws.

1.5 Key Findings and Contributions

The outcomes are incredible. Automated classification eradicated monotonous inbox tasks. The integration of the chatbot. Prioritizing features enhanced visibility of urgent messages and expedited decision-making processes. Visual charts and bars gave a simplified view of ocean of mails into small tank. Scalability indicated that the system could accommodate growth, catering to a firm tomorrow or a freelancer now.

This system encompasses functionalities beyond mere email management. It alters the methods employed in its vicinity.

1.6 Ethical Considerations

All research strictly followed ethical requirements. No human participants were involved. Also no personal information has been gathered. The research was

carried only after the completion of ethics training, the ethics application number is ETH2425-6900 and the certificate can be found in the appendices section.

1.7 Report Structure

The report is structured into seven main phases, akin to a design sprint.

- **Introduction:** Introduces the problem, the aims, the scope.
- **Literature Review:** An examination of attempted strategies, successful outcomes, and failures.
- **Methodology and Tools:** Explains the Methods and Tools carried out
- **Design and Development:** elaborates the design of the system and its architecture
- **Implementation and Testing:** Demonstrates the real time use of the methods and tools
- **Discussion and Critical Appraisal of Results:** Discusses findings and compares results
- **Conclusion:** Holds the limitations, future scope and progress

2 Literature Review

2.1 Integration of AI in Email Systems

AI integration in email systems is progressing quickly. NLP was altered by large language models (LLMs). Due to its bidirectional context understanding, BERT (Devlin et al., 2019) established a standard enabling summarization, classification, and spam detection. Then, models like LLaMA (Touvron et al., 2023) showed smaller and open-source structures which still well. performed. Ollama (Puhakka, 2025) makes these models deployable locally, which is important for privacy in email tasks.

On the other side, DeepSeek R1:1.5B (Guo et al., 2025) handles multi-turn, layered corporate emails better than LLaMA. Bigger models mean more semantic depth. Smaller models mean speed and resource efficiency. The trade-off is clear. This project work uses this trade-off by utilizing these models for real-time classification and a web application for summarisation and analysis.

2.2 Transformers

Transformers rudimentarily are architectures of neural built on self-attention enabling parallel, long-range sequence. Large language models (LLMs) scale transformers with massive data and compute to produce powerful, general-purpose text capabilities, but face efficiency, robustness, and alignment challenges.

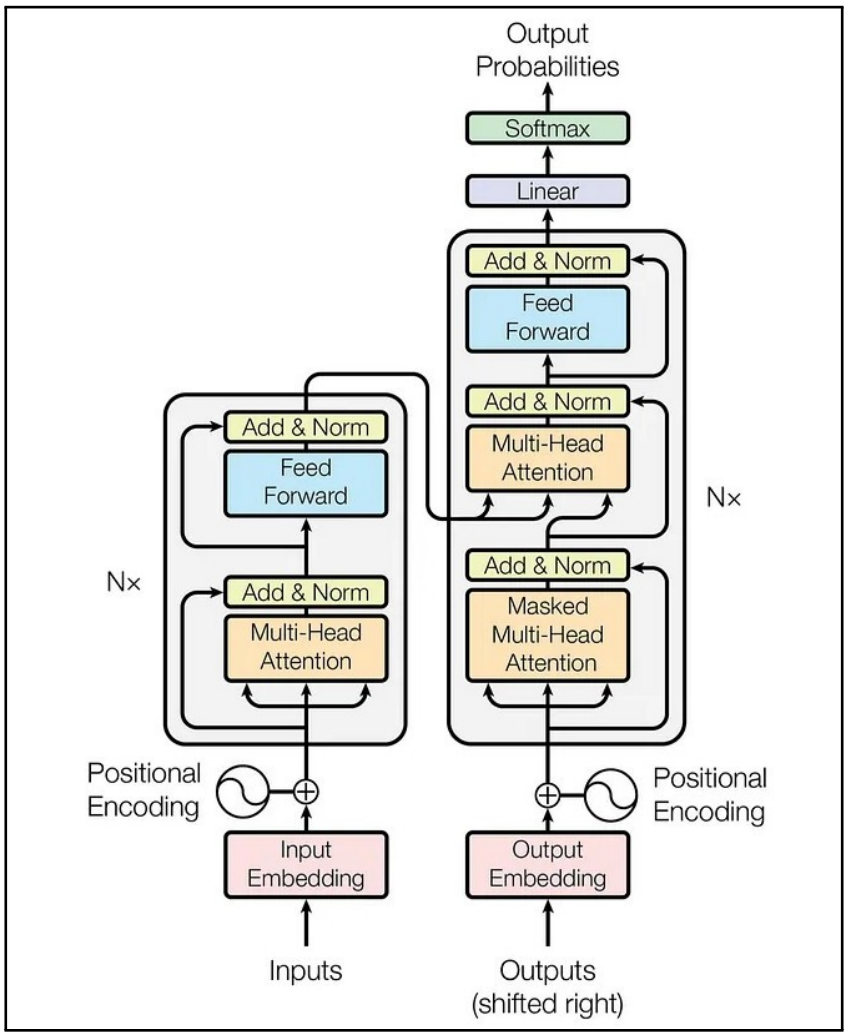


Figure 2.1: Transformers architecture (Medium, 2024)

2.2.1 Core architecture and attention mechanisms

Transformers replace recurrence with self-attention and position-aware token representations, enabling parallel processing and effective modeling of long-range dependencies. The attention mechanism evolved from earlier attention ideas and now includes multi-head attention, projection to queries/keys/values, and associated feed-forward, residual and normalization blocks (Luo et al., 2023), (Soydaner, 2022).

2.2.2 Transformers and attention — technical essentials

Self-attention computation Token embeddings are projected to queries, keys, and values; attention scores are scaled dot-products of queries and keys and used to weight values, enabling content-based interaction across positions (Luo et al., 2023). Multi-head attention Multiple attention heads run in parallel to capture diverse relational patterns; outputs are concatenated and linearly projected back to the model dimension (Luo et al., 2023). Position and depth Positional encodings or rotary embeddings supply order information; stacked layers of attention and MLPs produce hierarchical features (Luo et al., 2023). Specialized attention circuits Research identifies specialized attention heads (e.g., induction or selective induction heads) that implement copying, in-context pattern selection, and other algorithmic primitives inside transformers (Zheng et al., 2025). Variants for efficiency and bias Many efficient or biased attention variants (sparse windows, linear/causal approximations, structured scoring matrices) trade compute for locality or rank properties to handle long contexts or hardware constraints (Khan Raiaan et al., 2024), (Kuang et al., 2025), (Kemmer et al., 2025). Key interpretability notes: attention heads can implement distinct computational roles (copying, aggregation, positional tracking), so inspecting head circuits helps explain LLM behavior and failure modes (Zheng et al., 2025). BERT family Excels at representation learning for classification and retrieval because of bidirectional context modeling (Zaki, 2023). T5 and seq2seq Offer a unified text-to-text paradigm that simplifies diverse tasks into generation problems (Matarazzo and Torlone, 2025). Reviews and comparative studies document these distinctions and common adaptations used across tasks and domains (Khan Raiaan et al., 2024), (Zaki, 2023), (Matarazzo and Torlone, 2025). Latest architectural innovations and improvements Transformers continue to evolve with methods that improve adaptation, long-context handling, efficiency, and interpretability.Lately people been pushing new twists. things like adaptation tricks, mixing kinds of attention, scoring that has some structure, and even stacking layers in a more probabilistic way (Sun, Cetin and Tang, 2025), (Zhang et al., 2024), (Zahmad et al., 2024).

2.2.3 Main Ideas:

self-adaptive inference – Transformer^2 doesn't just run the same way every time. it kinda slips in these small expert vectors into the weights. so the model can shift its behavior on the fly, without doing the heavy full fine-tune (Sun, Cetin and Tang, 2025).

hardware-aware attention – GFormer is built with hardware in mind. it combines sparse and linear kernels, plus some windowed self-attention. that way it fits better with processors like Gaudi and can handle longer sequences more efficiently (Zhang et al., 2024).

probabilistic layer cascades – here the model doesn't just stack layers blindly. instead, it fuses info across layers in a probabilistic way. this helps cut down on error piling up and gives better flow for long-range stuff (Zahmad et al.,

2024). Semantic partitioning Hierarchical partitioning and contextual attention weighting allocate attention dynamically based on semantic relevance to reduce redundancy and improve long-context fidelity (Kemmer et al., 2025). Structured scoring matrices MLR and Block Tensor-Train scoring functions change the attention scoring inductive bias (full-rank or distance-dependent) to improve scaling and locality performance (Kuang et al., 2025). Emergent attention circuits Work on induction and selective induction heads explains how transformers discover and select causal structures in context, improving mechanistic interpretability (D'Angelo, Croce and Flammarion, 2025), (Choe et al., 2025). Each direction addresses a different bottleneck—adaptation cost, hardware efficiency, long-context coherence, or interpretability—and several approaches are complementary and integrable in modern pipelines (Sun, Cetin and Tang, 2025), (Zhang et al., 2024), (Zahmad et al., 2024), (Kemmer et al., 2025), (Kuang et al., 2025), (D'Angelo, Croce and Flammarion, 2025), (Choe et al., 2025).

2.2.4 Training methodologies

Training recipes and empirical scaling observations underlie LLM capabilities; both objective choice and compute/data scale determine emergent behaviors. Surveys synthesize common pretraining and adaptation pipelines and note predictable improvements from scale increases (Matarazzo and Torlone, 2025), (Shams, Salama and Callixtus, 2025).

2.2.5 Training paradigms and practices

Pretraining objectives Autoregressive next-token and masked-token objectives remain dominant; encoder-decoder objectives (text-to-text) unify tasks for flexible finetuning (Matarazzo and Torlone, 2025). Adaptation paths After base pretraining, models undergo task-specific fine-tuning, instruction tuning, or lightweight adaptation methods (LoRA-style adapters or dynamic weight edits) to align behavior and reduce compute needs (Matarazzo and Torlone, 2025), (Sun, Cetin and Tang, 2025). Efficient tuning Methods that modify low-rank components or add small adapters reduce downstream cost while retaining base model knowledge, and new self-adaptive approaches aim to adapt at inference without retraining (Sun, Cetin and Tang, 2025). Scaling laws Empirical studies show predictable improvements in loss and many downstream capabilities with increased model parameters, dataset size, and compute, but diminishing returns, data quality, and compute cost remain practical limits; certain architectural changes (e.g., MLR scoring) can alter empirical scaling behavior for specific tasks (Matarazzo and Torlone, 2025), (Kuang et al., 2025), (Shams, Salama and Callixtus, 2025). Empirical and theoretical scaling remains an active area: larger scale tends to unlock emergent abilities (chain-of-thought like behaviors and in-context learning), but record gains require matching data curation and training computation strategies documented in recent surveys (Matarazzo and Torlone, 2025), (Shams, Salama and Callixtus, 2025).

2.2.6 Applications, capabilities, current challenges and future directions

LLMs power translation, summarization, QA, code generation, decision support, and cross-disciplinary tasks, but they face robustness, hallucination, efficiency, and alignment challenges; active research priorities aim to mitigate these and extend LLMs into multimodal, modular, and interpretable systems (Matarazzo and Torlone, 2025), (Khan Raiaan et al., 2024), (Ding et al., 2025), (Shen et al., 2024).

2.2.7 Applications and strengths

Broad NLP and beyond LLMs are applied to translation, summarization, classification, planning, code generation, and domain-specific tasks in healthcare, finance, and science (Matarazzo and Torlone, 2025), (Khan Raiaan et al., 2024). Emergent reasoning At scale, models display in-context learning and emergent problem-solving behaviors useful for few-shot and zero-shot tasks (Matarazzo and Torlone, 2025).

2.2.8 Key challenges

Hallucination and factuality LLMs can produce non-factual outputs; architecture-level signals (intra-layer dispersion and inter-layer drift) have been used to detect hallucinations in a training-free manner (Ding et al., 2025). Robustness to adversarial or distributional shifts Dynamic attention and model-level defenses improve resistance to adversarial perturbations without heavy downstream costs (Shen et al., 2024). Compute and deployment costs Large models remain resource intensive; trade-offs between large and small LMs, and compression/distillation strategies, are crucial for practical deployment (Shams, Salama and Callixtus, 2025). Interpretability and mechanistic differences Different autoregressive designs encode facts and recall differently, so cross-architecture analysis is needed for reliable editing and safety (Choe et al., 2025). Mechanistic interpretability Mapping attention heads and circuits (induction/selective induction) to algorithmic functions to enable safer model editing and alignment (D'Angelo, Croce and Flammarion, 2025), (Zheng et al., 2025). Integration with retrieval and multimodality Augmenting LLMs with retrieval, external tools, and multimodal encoders to extend factual grounding and handle broader inputs is an accelerating research vector (Matarazzo and Torlone, 2025), (Khan Raiaan et al., 2024). Together, these directions aim to make LLMs more efficient, reliable, interpretable, and useful across domains; many recent papers propose complementary fixes that can be combined in production-grade systems (Sun, Cetin and Tang, 2025), (Zhang et al., 2024), (Zahmad et al., 2024), (Kemmer et al., 2025), (Kuang et al., 2025), (D'Angelo, Croce and Flammarion, 2025), (Ding et al., 2025), (Shen et al., 2024), (Choe et al., 2025).

2.3 Retrieval-Augmented Generation (RAG) in Email Intelligence

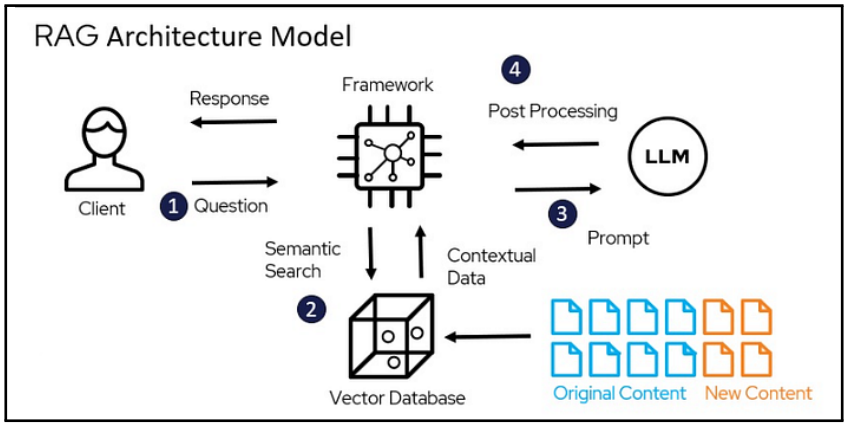


Figure 2.2: RAG architecture (Cubed, 2024)

Retrieval-Augmented Generation (RAG) basically bolts a retriever onto a big language model. idea is: grab outside info, cut down hallucinations, keep facts straight, and let the model update faster for new or niche stuff. started as just "retrieve then generate" pipelines. now you see dynamic versions, parametric bits, graph-based flows, even decentralized setups. people also building benchmarks, testing robustness, all that.

2.3.1 Development of RAG

RAG came in to patch some LLM issues — hallucinating, old knowledge, that kind of thing. it mixes outside text (from a corpus or index) with the model's own generation. surveys track the path: first, very simple retrieve-then-generate. later, more modular, more advanced RAG systems. these break things down into retriever, augmentation, and generator pieces (Gao et al., 2023), (Fan et al., 2024), (Cai et al., 2022). at first, folks leaned hard on dense retrievers and plugged in passages. later, the space blew up — taxonomies, clearer breakdowns of design choices. you get knobs like: what retriever, what kind of index, how to fuse stuff (Gao et al., 2023), (Cai et al., 2022).

- **Core idea**: append or fuse retrieved evidence into the LLM input (or its internal state) so generation is grounded in external text (Gao et al., 2023), (Fan et al., 2024), (Cai et al., 2022).

- **Motivations supported in the literature**: reduce hallucination, allow continuous knowledge updates, and support domain adaptation (Gao et al., 2023), (Fan et al., 2024).
- **Retriever role**: dense retrieval (DPR-like) became standard but has limits tied to pretraining and the retriever's exposure to facts (Reichman and Heck, 2024).

2.3.2 Key architectures and techniques

This section summarizes principal RAG architectures and representative techniques, connecting each to the surveyed literature and recent method families (Gao et al., 2023), (Fan et al., 2024), (Su et al., 2025a), (Ye et al., 2024).

- **Retriever improvement and analysis**: dense retriever training and limitations are analyzed to show retrieval is bounded by pretraining knowledge and decentralization of representations (Reichman and Heck, 2024).
- **Retrieval-aware fusion**: methods that pass retrieval metadata or embedding-level signals into generation (e.g., R$\hat{2}$AG) to reduce the semantic gap between LMs and retrievers (Ye et al., 2024).
- **Adaptive retrieval control**: autonomous iterative retrieval (Auto-RAG) and dynamic frameworks (DRAGIN) let LLMs plan and refine queries during generation (Tian, Zhang and Feng, 2024), (Su, Ai and Wu, 2024).
- **Parametric integration**: Parametric RAG directly encodes documents into FFN parameters to reduce runtime context size and improve efficiency (Su et al., 2025a), (Su et al., 2025b).
- **Robustness modules**: corrective and evaluative layers (CRAG, retrieval evaluators, RPO) assess retrieval quality and trigger corrective actions such as web augmentation or preference-aware alignment (Yan et al., 2024), (Yan and Ling, 2025).

2.3.3 Key comparative observations supported by experiments and benchmarks:

- **Retrieval quality matters more than sheer context size**; exact retrieval can reduce end-to-end cost by sending fewer higher-quality docs to the generator (Quinn et al., 2024).
- **Dynamic/iterative schemes** (Auto-RAG, DRAGIN) improve information integration but trade off latency and require reliable stopping criteria (Tian, Zhang and Feng, 2024), (Su, Ai and Wu, 2024).

- **Graph and distributed variants** address domain reasoning and privacy/scalability respectively, but at the expense of infrastructure complexity (Zhang et al., 2025), (Xu et al., 2025).

2.3.4 Applications, benchmarks, advances and evaluation

This section connects RAG use cases, the evaluation ecosystem (2020–2025), and notable innovations while citing benchmark and application papers (Chen et al., 2023), (Lyu et al., 2024), (Quinn et al., 2024), (Jiao et al., 2024).

2.3.5 RAG applications and use cases are broad and include:

- **Open-domain and multi-hop QA** (HotPot-type tasks) where retrieval supplies multi-document evidence (Jiao et al., 2024).
- **Knowledge-intensive generation** such as summarization, update/patching of texts, and domain assistants (e.g., healthcare case studies) documented in applied reports and surveys (Gao et al., 2023), (Fan et al., 2024).

2.3.6 Benchmarks and metrics

- **RGB benchmark**: decomposes RAG evaluation into testbeds for noise robustness, negative rejection, information integration, and counterfactual robustness to evaluate how LLMs use retrieved evidence (Chen et al., 2023).
- **CRUD-RAG**: expands evaluation beyond QA into Create/Read/Update/Delete categories and stresses retriever, context length and KB construction effects (Lyu et al., 2024).
- **Common metrics used**: retrieval recall/precision, end-task accuracy (EM/F1 for QA), ROUGE/BLEU for summarization, semantic answer similarity (SAS) for descriptive QA, plus system metrics (latency, throughput) for deployment (Lyu et al., 2024), (Quinn et al., 2024).
- **Hardware and latency benchmarks**: work on accelerating exact nearest-neighbor search (IKS) shows that faster exact retrieval can lower end-to-end inference time substantially, affecting system-level evaluation (Quinn et al., 2024).

2.3.7 Recent advances and representative innovations (2020–2025)

- **Iterative/autonomous retrieval**: Auto-RAG enables LLM-driven multi-turn retriever dialogues to plan and refine retrievals, improving performance on complex queries (Tian, Zhang and Feng, 2024).
- **Dynamic retrieval frameworks**: DRAGIN adapts retrieval to the LLM's information needs and proposes lightweight, information-driven retrieval policies (Su, Ai and Wu, 2024).
- **Parametric RAG**: injecting document knowledge into model parameters (Parametric RAG) reduces runtime context and can be combined with in-context RAG for gains (Su et al., 2025a), (Su et al., 2025b).

2.3.8 Future directions

This section describes remaining obstacles, measurement gaps, and prioritized research directions tied to the cited literature and benchmarks (Chen et al., 2023), (Zou et al., 2025), (Yan et al., 2024), (Yan and Ling, 2025), (Quinn et al., 2024).

Opening summary: RAG systems face challenges in retrieval quality, robustness to poisoned or contradictory sources, latency/scalability trade-offs, and evaluation gaps; the literature identifies defensive, algorithmic, and infrastructure research vectors to address these issues (Chen et al., 2023), (Zou et al., 2025), (Quinn et al., 2024).

2.3.9 Challenges and limitations

- **Retriever-generator misalignment**: LLMs and retrievers differ in objectives, producing a semantic gap that harms integration unless retrieval info is incorporated explicitly (Ye et al., 2024).
- **Robustness to bad or adversarial knowledge**: poisoning and corrupted KB attacks can mislead RAG outputs, requiring provenance and integrity checks (Zou et al., 2025).
- **Evaluation coverage**: existing benchmarks often concentrate on QA; RGB and CRUD-RAG expand coverage but more multi-task, multilingual, and real-world evaluation is needed (Chen et al., 2023), (Lyu et al., 2024).
- **latency vs fidelity** – if you do iterative retrieval or exact retrieval, you get better facts. but it costs more time and compute. some of this pain gets eased by hardware accel or using parametric tricks (Tian, Zhang and Feng, 2024), (Quinn et al., 2024), (Su et al., 2025a).

maintenance mess – parametric setups and graph-based designs aren't free. they need careful update flows and real cost tracking (Su et al., 2025a), (Zhang et al., 2025).

2.3.10 Future considerations

Retriever-aware training – train LLMs not just on answers but on how good the retrieval was. tie it into reward models, aligners, things like RPO. goal: model learns how to trust or downplay external text (Yan and Ling, 2025).

Hybrid systems – mix parametric doc injection with lighter in-context augmentation. idea: balance speed, freshness, and coverage (Su et al., 2025a), (Su et al., 2025b).

Provenance and safety – gotta know where a source came from. plus detect poisoned entries in KBs. and have fallback paths if retrieval looks shady. attacks already shown this matters (Zou et al., 2025).

- **Multimodal, graph and decentralized retrieval**: extend RAG to structured graph knowledge for multihop reasoning and to decentralized architectures for privacy and edge deployment (Zhang et al., 2025), (Xu et al., 2025).

- **Benchmarking diversity**: develop evaluation suites that jointly measure retrieval quality, generation grounding, robustness, and system costs across languages and domains (Chen et al., 2023), (Lyu et al., 2024), (Quinn et al., 2024).

  LLMs are powerful but stuck with static training data. Updating is hard. RAG (Lewis et al., 2021) mixes an LLM with retrieval from an external corpus. It pulls relevant documents and feeds them back to improve generation. In email, this is critical. Summaries and replies can be factually tied to real content instead of hallucinations. Studies on the Enron Email Corpus (Klimt and Yang, 2004) show improved factuality. But most of these experiments remain limited to old, English-only corpora.

  This system uses RAG pipelines directly in the web interface. Emails are embedded, indexed, retrieved, and grounded into responses.

2.4 Frameworks and Tools

A pipeline structure of loaders made of Retrievers and LLMs was presented by LangChain (Topsakal and Akinci, 2023). When mails go beyond limits, chunking gets involved. Jiang, Ma, and Chen (2024) demonstrate how chunking strategies' overlap and length impact the final product. Embeddings in FAISS or Chroma (Johnson, Douze and Jegou, 2017) make fast similarity search possible, critical in real-time cases.

Deployment tools matter. Streamlit (Singh, 2021) allows developers to build dashboards quicker. This has influenced the project work. Vector databases and embeddings supported retrieval in system. The web app handles summarisation and analytics. The browser extension does inbox classification.

2.5 Historical and Theoretical Aspects on Email Management

Email began in the ARPANET era (Cerf & Kahn, 1974). Earlier, systems were text-based. Next came arrival of folders, filters, attachments (Klahr, 2015). Growth in usage created email overload. Karahanna and Selnes (2003) describe overload as stress from too many irrelevant messages. Harrison et al. (2010) link it to stress, low productivity, and moreover burnout. Research includes prioritization and automation (Tang & Rosson, 2009).

This history shows why better tools are always needed. It emphasizes why AI in email is more of a necessity.

2.6 Automation in Email Systems

Automation were there before arrival of LLMs as spam filters were functioning on relied rules or ML (Baeza-Yates & Ribeiro-Neto, 1999). Later came auto-replies, scheduling, and classification (Klahr, 2015). Chatbots appeared too. Liu et al. (2016) and Kim & Choi (2018) show bots can interact with users, manage inboxes, even compose responses. But they fall short when language or context is complex. Errors happen. Adaptability is weak.

This artefact directly targets these weaknesses. It combines automation with context-aware, retrieval-driven LLMs. That balance makes responses both fast and grounded.

2.7 Research Gaps and Positioning of this Work

Several gaps remain:

- Performance comparison between lightweight large language models in email tasks is missing. LLaMA vs DeepSeek in mail dataset has not been tested on the functionalities of summaries, generating charts, classification, prioritization.
- Static datasets like Enron dominate experiments. They are outdated, not multilingual, not dynamic.
- Chunking and embedding database choices are underexplored in terms of real quality retrieval.

This project sits inside these gaps. It utilizes llm models for different layers of email tasks. It performs context based retrieval in practice. It also delivers a web application for analysis. This project focuses more on innovation and performs research smaller dataset, Dense retrieval with InMemory vector store and lightweight embedding model like llama3.2 and DeepSeek were considered for efficient choice. It provides low latency, more simplicity and gives flexibility without infrastructure and computation costs.

3 Methodology and Tools

This section delves into tools and techniques utilized to built the Email Management System (EMS). It also covers the approach that helped us get there. The idea is provide breakdown of comprehensive understanding of the strategies and technological components and how it all fits together.

3.1 Methodology

Retrieval-Augmented Generation (RAG) pipeline was the top recipe in compiling this project. Other options were—like fine-tuning a model ML classification—but RAG checked the most of the boxes.

3.1.1 Problem Definition and Objectives

The primary goal was to build a system that read structured file and language model to understand and simplify data and convert to more valuable insights. Also let users ask questions in plain language through a chatbot.

3.1.2 Data Aquisition and Preprocessing

Initially the project was started with public email dataset which was in the form of plain text file. Cleaned it a bit. Removed missing stuff, trimmed useless parts. After cleanup, data was chunked and sent to the LLM. It needed clean input to give good output.

3.1.3 Data Extraction using LLM and RAG

LLM from Ollama has been used to pull key info. Directly prompted summaries. When it wasn't enough, used RAG — it fetched extra info from outside. This helped the model in giving better responses.

3.1.4 Reason behind RAG

- **Accuracy:** RAG helps cut down on hallucinations by making the model refer to real emails data.

- **Faster:** Fine-tuning is expensive and takes a lot of compute. RAG skips all that by using a pre-trained model and just pulling in relevant context.
- **Easy Updation:** If you get more emails, just drop them in the folder. Retraining was not required. Much simpler than repeatedly creating a fine-tuned model.

3.1.5 Justification

- **Fine-tuning:** Needs a big labeled dataset. We didn't have that. Plus, it's slow, expensive, and has privacy risks.
- **Keyword Search (e.g. grep):** Way too basic. It doesn't understand what you *mean*—just whether the word is there. Useless if the language doesn't match exactly.
- **Limitations of RAG:** It's not perfect. If the retrieval doesn't find the right stuff, the model gives a wrong answer anyway. Also, any built-in bias from the LLM we used (Ollama) is still there.

3.1.6 Prompt Engineering

Prompts played major role. Poor ones gave bad results. Spent time testing short, clear ones. Tweaked wording to match the structure of the data. This made the model more reliable.

3.1.7 System Workflow with LangChain

LangChain connects everything. It passed user queries to the LLM, then kicks off a similarity search using an in-memory vector store if RAG gets triggered. After that, it handles the output and sends it where it needs to go. The chatbot and chart tool are both hooked into it.

3.1.8 Chart Visualization

Charts were generated by Plotly Express. It was simple and fast. After fetching results from llm, they are converted to json objects, shaped them into a format Plotly could handle. Made bar, line, pie charts depending on the type of question. Charts were shown in the UI.

3.1.9 Chatbot

The chatbot was implemented for users to talk to mails through the system. They type stuff like “show me top mails” or “what did john mailed me.” Once the query comes in, it gets passed to the backend. The system grabs the data, makes a chart, and throws back the answer. All of it sits inside a Streamlit app. I built the whole thing there — upload button, text box, response area — didn’t need to mess with front-end code.

3.1.10 Testing and Evaluation

Each part was tested on its own — file reading, the model’s output, the charts, chatbot replies. Then full application was ran from start to end. Restarted several times and fixed bugs.Sometimes the bot made no sense. Other times, the chart didn’t match the numbers. Once again check was carried out, tweak, test again. improving bit by bit.

3.2 Tools

Below are the tools used and the reason behing their usage.

3.2.1 Python

- **Definition:** Modern programming language extensive support of AI related libraries and frameworks.
- **Pupose:** To built entire backend and connect.
- **Application:** It is the base and platform for orchestrating all other tools

3.2.2 ‘os‘

- **Definition:** Standard Python module which acts as shelf organizer in handling files from directories.
- **Pupose:** It is used for count on parsing files from directories.
- **Application:** We use it to cruise through folder aisles, spot ‘.pdf‘ and ‘.eml‘ items, and grab their full file paths for checkout. Super handy.

3.2.3 LangChain (Community + Core)

- **Definition:** It is like a conveyor belt that chains llm operations.
- **Pupose:** It automates the messy parts like pulling documents, generating embeddings, building prompts, and managing LLMs—like a well-oiled machine at the backend of a smart warehouse.
- **Application:** Every stage of the RAG pipeline runs through this. From document prep to final output, LangChain keeps the system humming with minimal wiring.

3.2.4 ‘OllamaEmbeddings‘

- **Definition:** It scans the meaning of the text.
- **Pupose:** It runs locally, keeps everything private.
- **Application:** Each email or doc chunk gets tossed into this module. Out comes a high-dimensional vector that tells us what it really means, not just what it says.

3.2.5 Ollama

- **Definition:** The tool serving as connecting agent between llm and application.
- **Purpose:** It's the platform that runs generative brain. we may feed a question and some context, and it speaks back like it knows what it's doing—because it does.
- **Application:** Final stage in the pipeline. It looks at the question, grabs the retrieved chunks, and produces a coherent, context-aware answer.

3.2.6 ‘RecursiveCharacterTextSplitter‘

- **Definition:** A smart slicer for text.
- **Pupose:** Instead of just chunking by size, it looks for natural breaks like paragraphs, sentences, also line spacing so that chunks won't get cut in the middle.
- **Application:** We throw full emails or PDFs in, configure the ideal chunk size and overlap, and out come nicely portioned text bits ready for embedding.

3.2.7 ‘InMemoryVectorStore‘

- **Definition:** RAM-based storage to store and index generated text embeddings for similarity search and retrieval.
- **Pupose:** It cuts need for big warehouse (like Chroma or Weaviate) still keeping everything accessible due to it's in-memory.
- **Application:** Stores the semantic vectors plus and their original chunks. When query is supplied, it finds relevant "ingredients" in matter of time.

3.2.8 ChatPromptTemplate‘

- **Definition:** It is a format card with different inputs each time.
- **Pupose:** Helps us separate instructions (prompt logic) from the groceries (user input). You don't want to re-write the recipe every time, just swap out the context and question.
- **Application:** We set up a system message (telling the AI what kind of chef it is) and a user prompt with placeholders. Pop in the ingredients, and the LLM understands what to return.

3.2.9 JsonOutputParser‘

- **Definition:** It is a Box labeler.
- **Pupose:** This formats output to machine-readable. Easier for other system parts to process, move on.
- **Application:** Enables LLM output into a strict JSON structure like:
  “‘json "person": "john", "mails": [2]

3.2.10 Other Utilities

- **‘email‘:** It cuts ‘.eml‘ files and unpacks body and meta information
- **‘re‘:** Like a barcode detector. Spots patterns in messy text and helps clean it up—removing junk, stray whitespace, and weird formatting in this project.
- **‘itertools‘:** It is Perfect conveyor belt for efficiently looping through big lists of chunks without loading the whole warehouse into memory.
- **‘pandas‘:** The spreadsheet aisle. For anything involving tables—metrics, logs, evaluation data—this is where it all goes.

- **‘plotly.express‘:** The storefront window. It visualizes given analysis. generates interactive graphs.
- **‘json‘:** This Encodes data for transport and storage. Used in configs, LLM responses, which then fed for visualizations purpose.

4 Design and Development

This whole part elaborates key components on process of building the app—the pieces of ehich consists of and where they fit, and role of tools. Stuff had to work smooth, feel easy, and not crash when fed 500 emails at once. Here's how we did it.

4.1 System Architecture

System is built in a way shown in the below figure. Each block handles particular job: crunching emails, preprocessing and building graphs. Why? Easier to fix, easier to scale, way less headache later. Drop in new tools anytime—no full rework needed. Keeps everything snappy, clean, safe.

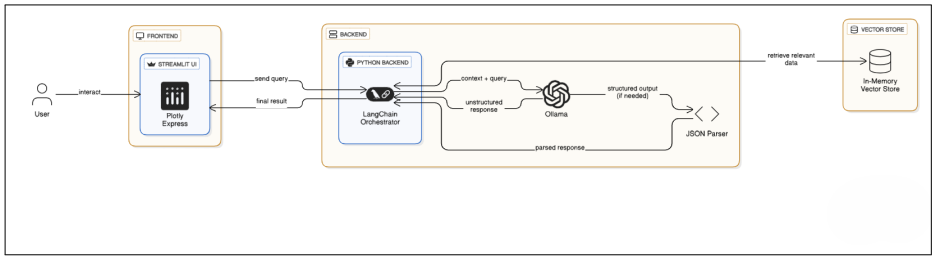


Figure 4.1: Overall Architecture

When emails are loaded, the system doesn't just drop it in inbox. Instead, it immediately pulls out the useful pieces: who sent it, the subject line, the body, and the timestamp. With that information in hand, it starts doing some smart processing. The email can be summarized so we don't have to read the whole thing, analyzed to understand the content, categorized into the right group, and even prioritized so that we know which ones deserve better attention first.

From there, the system turns all that work into something easy to use — a short summary, a visualization if needed, a classification of the type of email, and even a priority list to help focus. These results are then displayed in a clean interface so you're not buried in raw text.

At the same time, we can interact with the system through a chatbot. Instead of hunting through emails all ourselves, we could just ask questions like "What are my most important emails today?" The chatbot fetches the data, processes it, and gives you an instant response right in the chat.

In short, the whole flow takes messy, time-consuming email reading and transforms it into a clear, organized experience — whether you want to glance at summaries, see patterns, or simply ask the chatbot to do the work for you.

4.2 Frontend

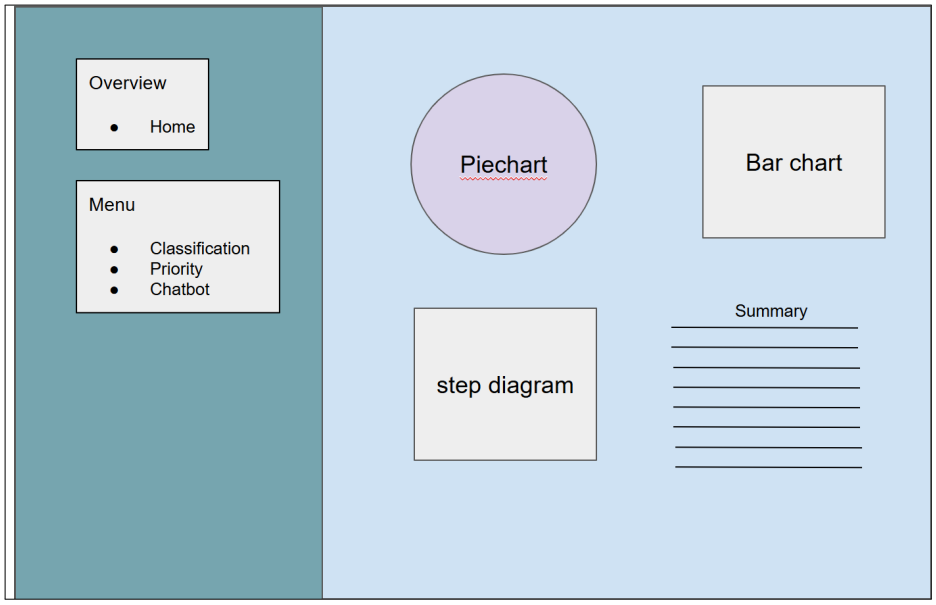


Figure 4.2: Frontend Design

This is the user face. Built with Streamlit—fast, Python-native with no heavy frontend mess. Few key bits:

- **Left Navigation bar:** Constitutes of two sections Overview and Menu. Overview contains the Home navigation which has dashboard like UI and summary of mails and the menu presents three navigation points such as classification, priority and Chatbot
- **Dashboard:** Shows analysis—how many people filled what portion of inbox in a piechart. Categories in a bar chart, Priorities in step diagram and finally summary as text.

4.3 Backend and Vector store

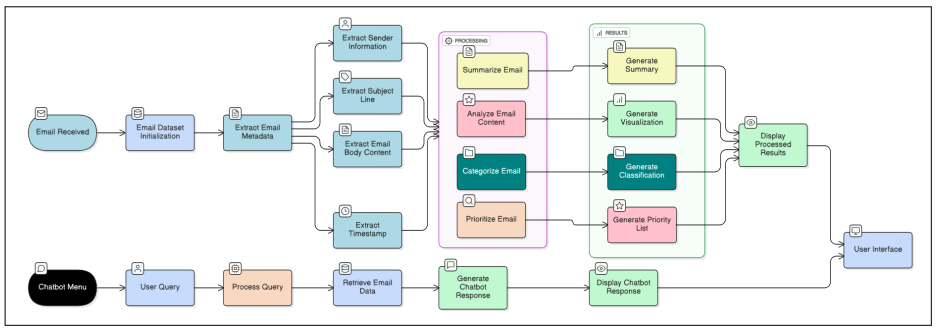


Figure 4.3: Backend Architecture

4.3.1 System Flow

It splits in two paths one scans and sorts and the other answers questions. That's how this thing runs.

4.3.2 Dataset Preprocessing

Emails dropped in one spot. System scans them to pickup, preps for parsing. Chunk→ Index → ready.

4.3.3 Metadata Pull

Grab top-level info—like labels on the package.

- Sender name
- Subject line
- Body content
- Timestamps This becomes the base for all next steps.

4.3.4 Processing

chunks the mails to feed llm and runs deep analysis:

- Summarize: Generates a gist of collective mails
- Analyze: Looks for person and number of mails sent.
- Categorize: Work? Spam? Business?Personal?
- Prioritize: Flags stuff needing fast response.

4.3.5 Results

- Summary: Displays Summarized text
- Charts: Who sends most mail?
- Categories: Work, Personal, Business, Social.
- Priority list: What needs eyes *now*.

  All that → passed to display.

4.3.6 Chatbot path

Now when a user wants to ask questions like

- What did John say ?
- "Any urgent emails today?"

Bot flow:

- Start: User clicks to input query.
- Parse: Bot parses the query and read it..
- Search: Digs up right email bits.
- Respond: Generates chat-style kind of relevant and useful response.

Example:

- 3 emails from John and urgent saying to shedule a meet asap.

4.3.7 Final Interface – One Place for All

UI's the meeting point. Combines:

- Auto stuff: Charts, lists, summaries.
- Chatbot stuff: Q&A, custom replies.

  User can go:

- Passive: Just browse what's already shown.
- Active: Ask stuff, get dynamic answers.

  Top flow = machine-led. Bottom flow = user-led. Both end at same place—clean, unified interface.

5 Implementation and Testing

This part shows how design turned into code. Real thing, running. First, see what the app looks like, then dive into guts of code, then hit testing—did it work, did it break, how strong it is. Whole deal is about an email system that loads mail, lets us ask stuff in plain words, spits answers back.

5.1 Implementaion

5.1.1 Application Overview

Overall UI consists of four main components namely Home, Classification, Priority and chatbot. User can switch between components through a left navigation menu bar.

Home Component

The home component is the default one that shows up on running the app. It consists of home page that displays three charts and one text portion. The first one is a pie chart which most contributed mails of particular person in percentages. The next one is a bar chart which represents the classification part.

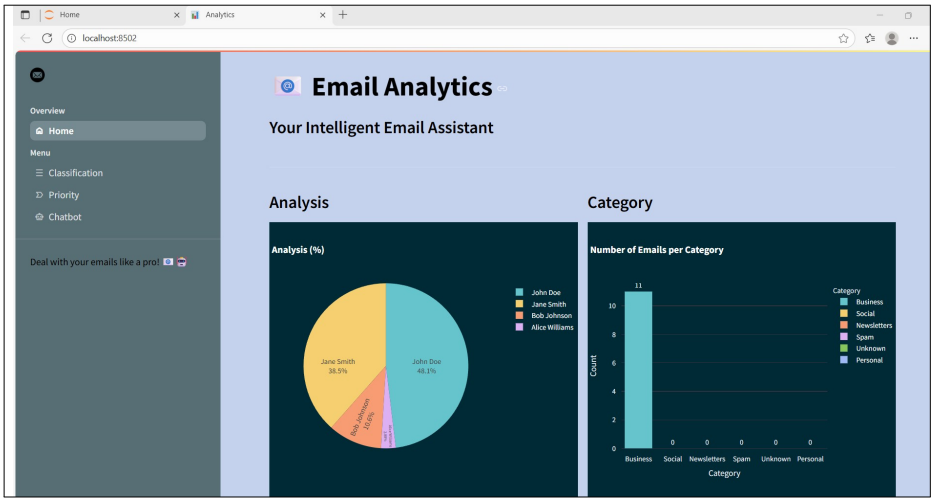


Figure 5.1: UI Home component 1

The below diagram is the continuation that holds a funnel chart for presenting the priority and quantity of mails in each. Final portion is the Summary of the mails.

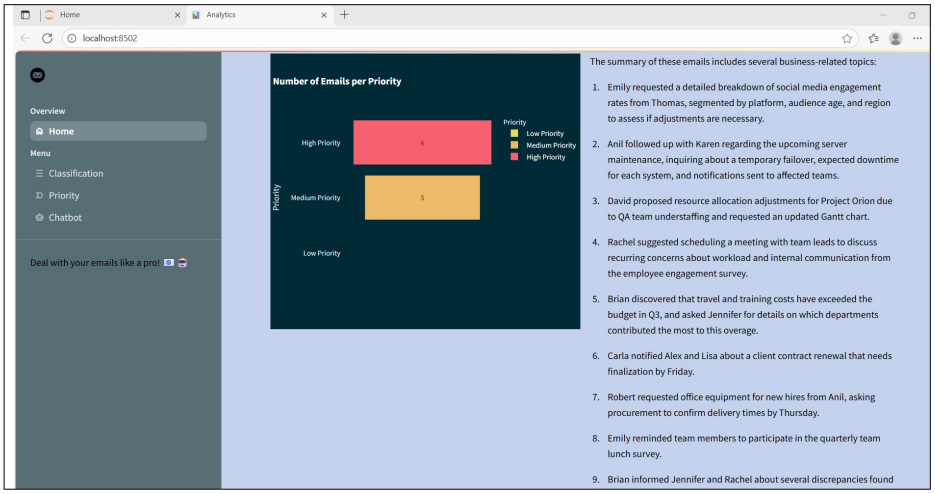


Figure 5.2: UI Home component 2

Classification Component

A person can switch to this component through the navigation bar, this components hold the mails under defined boxes of categories. The Defined categories are Business, Social, Newsletters, Spam, Unknown and Personal. When the user

selects categories the mails classified under that shows up. These categories are dynamic and generated based on llm's response thus they appear only if there are mails under that, as we can see in the below figure only business category is displayed and others are vanished due absence of mails in that.

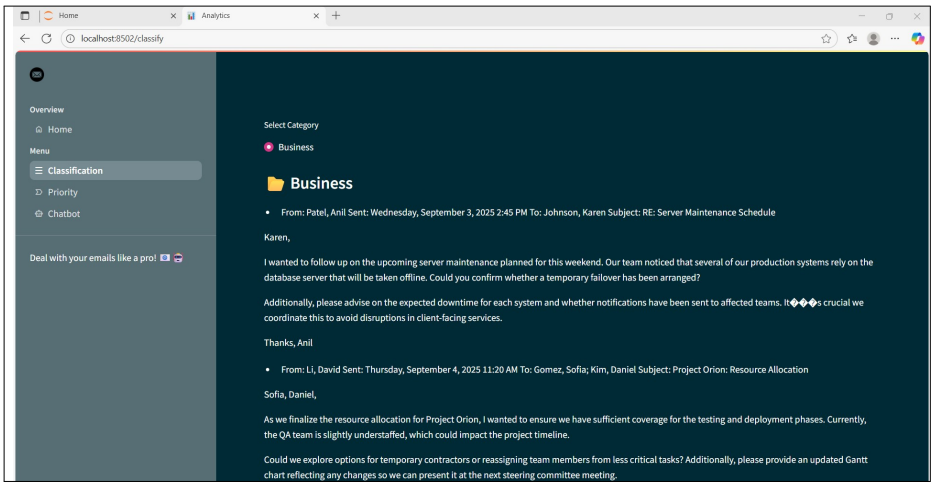


Figure 5.3: UI Classification component

Priority Component

This component has the mails sorted and grouped based on priorities. This group has three elements - High Priority, Medium Priority and Low Priority. They are dynamic too which means they appear only when mails are present in that.

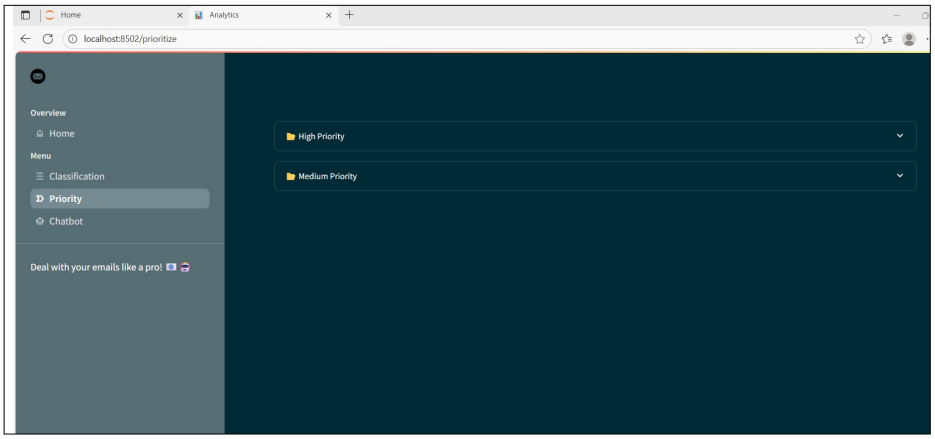


Figure 5.4: UI Priority component 1

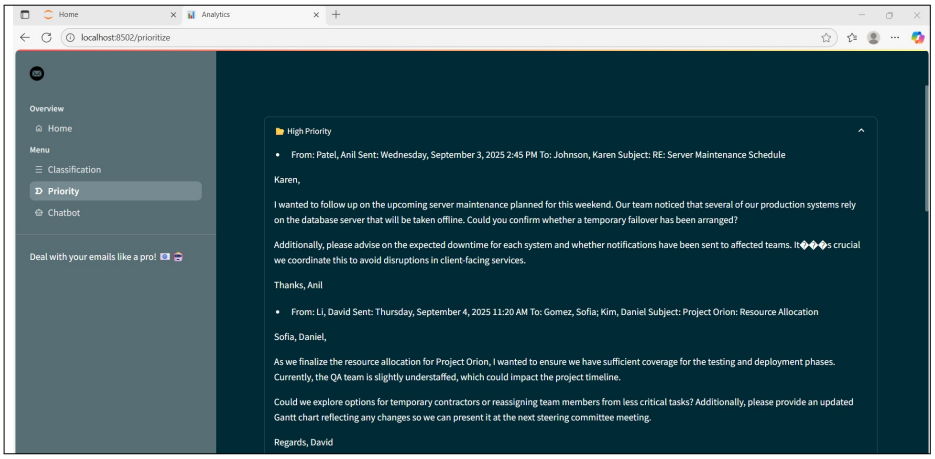


Figure 5.5: UI Priority component 2

Chatbot Component

Chatbot is a final component that interacts with the user. Here user can ask queries regarding the mail and get answers and the chat history stays even after switching pages.

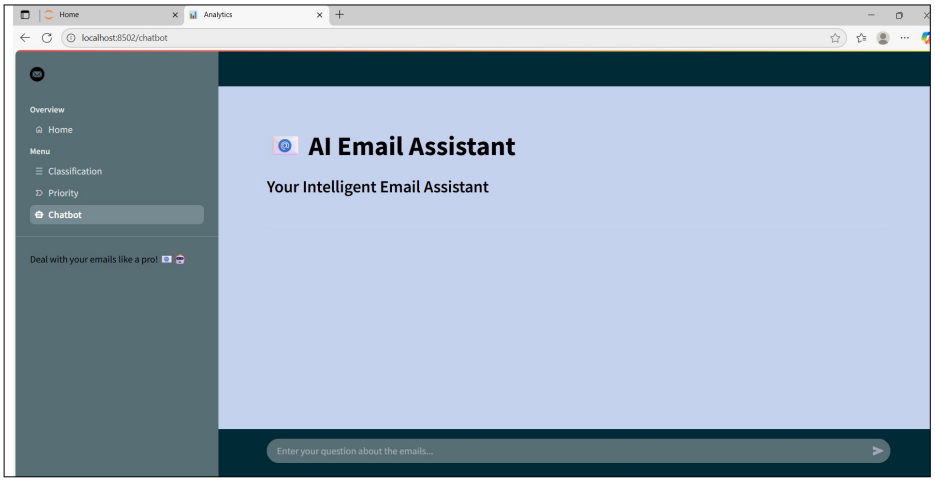


Figure 5.6: UI Chatbot component 1

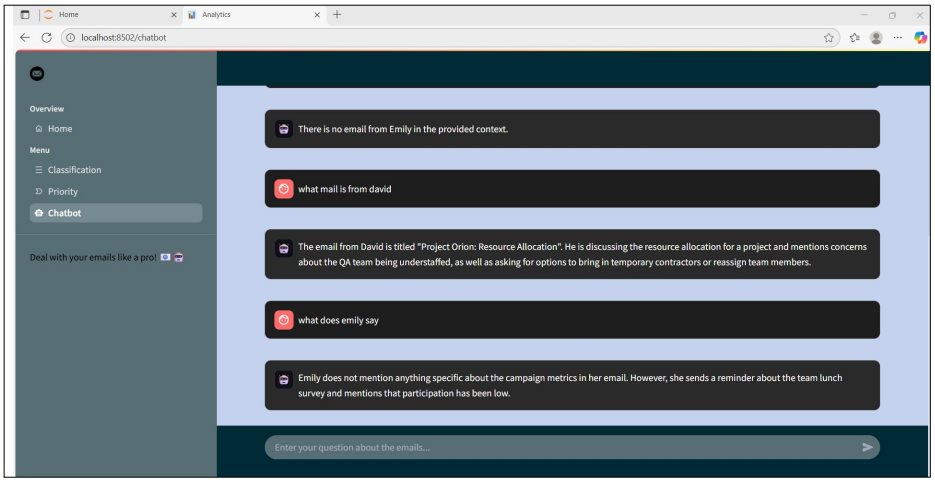


Figure 5.7: UI Chatbot component 2

End of design path → CLI app. Load folder of emails. Type query. System parses, indexes, talks back JSON. Flow goes: Load → Process → Query → Response. End goal → app that scales, stays clean, doesn't leak data, doesn't choke on volume. Backend = Python. Handles NLP, stocks, parsing. Frontend CLI → simple, quick.

5.1.2 Datasets

This project involves two types of datasets one is public dataset (Cohen, 2015) and the other is synthetic dataset (Author's Own, 2025) which was created for better testing and results.

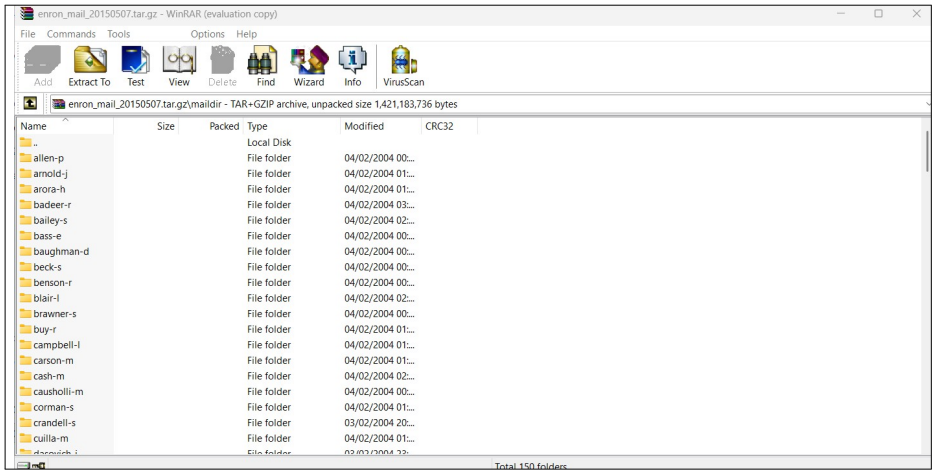


Figure 5.8: Public Dataset 1

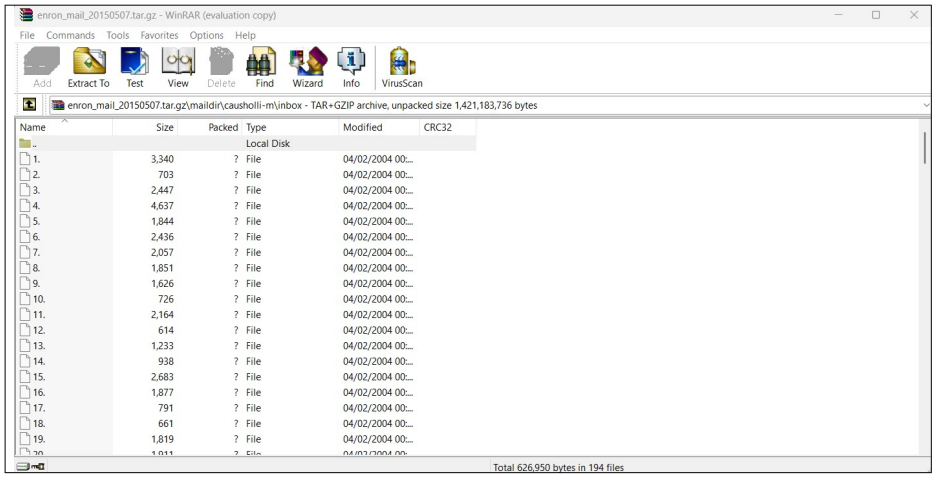


Figure 5.9: Public Dataset 2

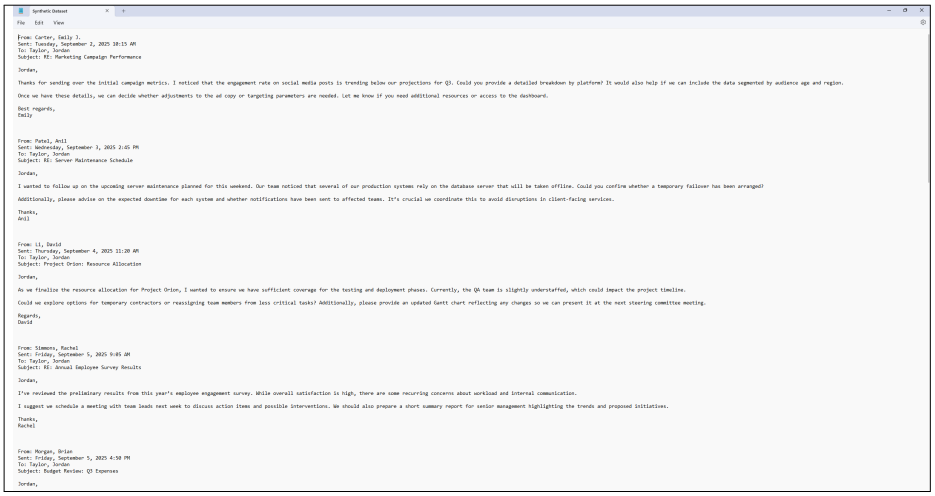


Figure 5.10: Sythetic Dataset 1

5.1.3 Overall Code

Project Structure

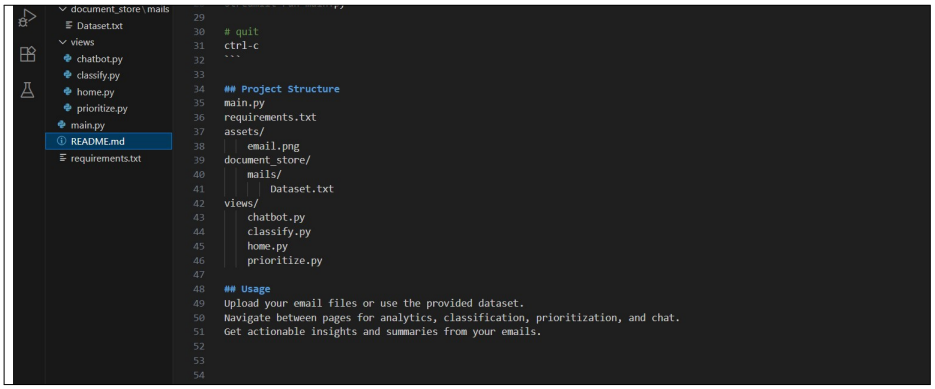


Figure 5.11: Project Structure

The project structure consists of four major parts which then holds further sub parts. Lets look into the breakdown of each part.

- **'main.py'** This is the engine of the entire application. It's the main script we run to start the application. It ties all the different pages and functionalities together into one seamless web app.
- **'requirements.txt'** This is the list for all the ingredients (Python libraries like Streamlit, Pandas etc.) needed to make the application work.

Before we start, we could use this file to install everything the project depends on.

- **‘assets/‘ & ‘document_store/‘** These are filing cabinets.
- The ‘assets/‘ folder holds static files like images (e.g., ‘email.png‘ for the UI icon).
- The ‘document_store/‘ folder, specifically the ‘mails/‘ subfolder, is where we feed email files for the app to analyze. The ‘Dataset.txt‘ file is likely a sample or pre-loaded set of emails for us to experiment with.
- **‘views/‘** This folder contains the different rooms or pages of the application. Each file is a dedicated page with a specific purpose:
- **‘home.py‘**: The front door or dashboard. This is the first page we see, giving us an overview and navigation to the other sections.
- **‘classify.py‘**: The organizer. This page automatically categorizes our emails into types like "Promotions," "Social," "Work," etc.
- **‘prioritize.py‘**: The personal assistant. This page analyzes our emails to figure out which ones are most important and urgent, helping us focus on what matters first.
- **‘chatbot.py‘**: The smart search bar. This is where we can have a conversation with our data. Ask questions like "What did Sarah say about the project deadline?" and get instant answers based on our emails.

In short, on starting the engine (‘main.py‘), it runs and lets us navigate through different views (‘views/‘) to analyze, organize, and chat with the emails we've stored (‘document_store/‘).

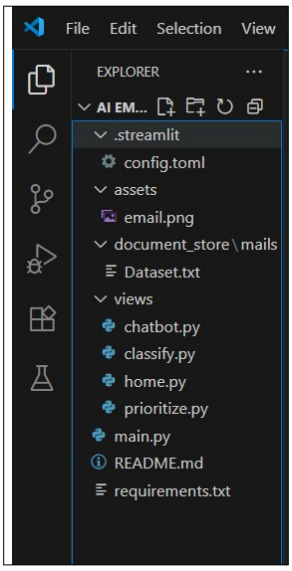


Figure 5.12: Folder Structure

Main Component

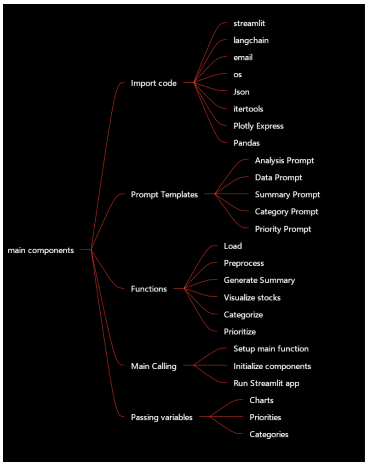


Figure 5.13: main.py Structure

We could say that main.py is the brain and command centre of all we do. We put everything together here to make a strong and unified app. Here's the setup:

Importing Section

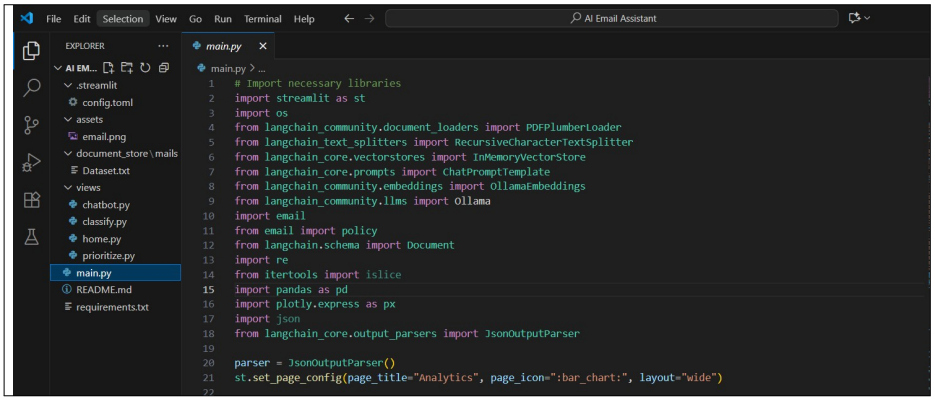


Figure 5.14: main.py component 1

First, we get all the strong libraries and frameworks that do the hard work for us. This is what we have to work with: streamlit: It is the base of our web app and lets us make the interactive interface. langchain: The AI powerhouse that takes care of our sophisticated language processing and links prompts and LLM (Large Language Model) calls. Email, OS, and JSON are our tools for working with email files, navigating the operating system, and working with data. pandas: Our data wrangler, it puts data into nice tables so we can easily look at it. plotly.express: The artist that makes the gorgeous and interactive charts that show our findings. itertools: A tool that helps you work with sequences of data quickly.

The initial code set ups libraries and modules like ‘os‘, ‘re‘ and ‘json‘ to work with files in parsing and reading them. The ‘streamlit as st‘ runs UI. Rest of stack has ‘plotly.express‘ and ‘pandas‘ for charts to represent numeral data. On the model side, ‘ollama‘ runs llm locallly and coonects app to perform task like summarizing, generating analysis, classification etc.

The variable ‘SAVE_PATH‘ points to parent folder of email dataset files, so that sessions can continue and hold between runs.

The Templates for Prompts

Prompt templates keep things in line. They give shape. They stop drift. Tone stays the same. Instructions don’t get fuzzy. Output? Cleaner.

The main ones are.

- **Data prompt template** → Instructs to convert analized data to single json object.
- **Analysis prompt template** → Instructs AI to look at an email and pull out information like sender, mails and percentage.

- **Classification prompt template** → drops emails pre-set category buckets: work, personal, business, spam and unknown.
- **Priority prompt template** → stares at sender, subject, content. Chooses amoung high priority, medium and low, figuring out how important and urgent an email is.
- **Chatbot prompt template** → lets a question come in, spits an answer out.
- **Summary Prompt** → Tells AI to make short, useful summaries of collective emails.

These prompts cut friction. Keep style. Keep structure. But they bend too. Adjust over time. Input shapes them. Behavior shifts them. Models learn, replies sharpen. Words change, reliability rises. It's all iteration. Always moving. We don't just tell the AI what to do in general terms; we write clear, specific instructions, or "prompts," to make sure the outputs are always the same and of high quality.

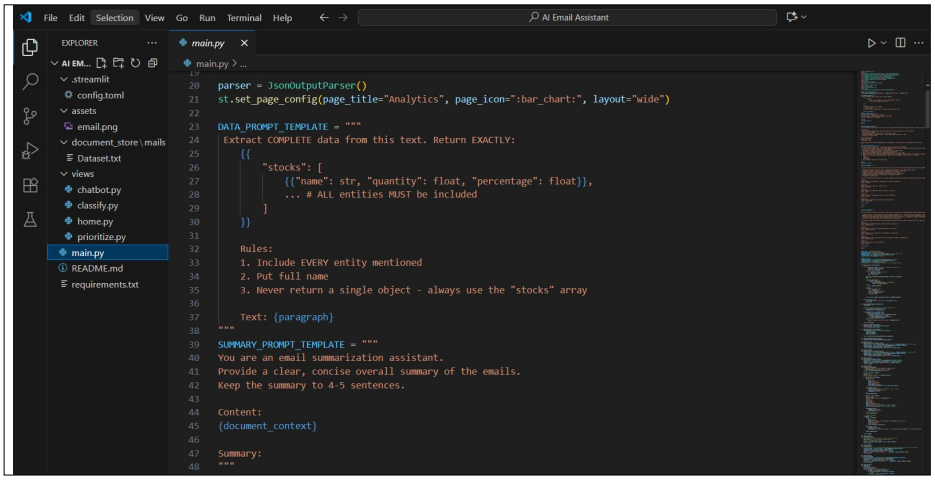


Figure 5.15: main.py component 2

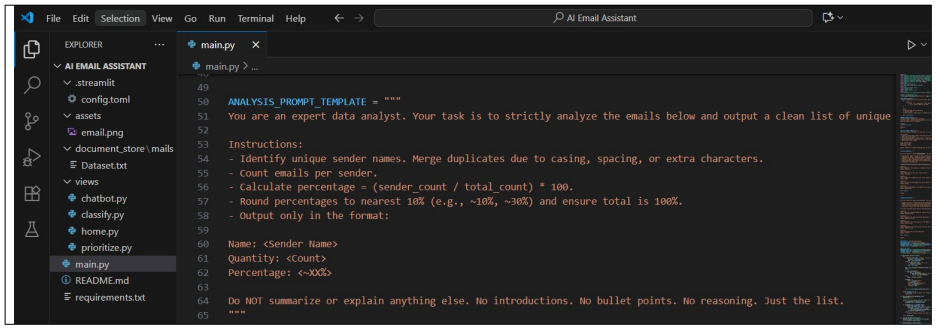


Figure 5.16: main.py component 3

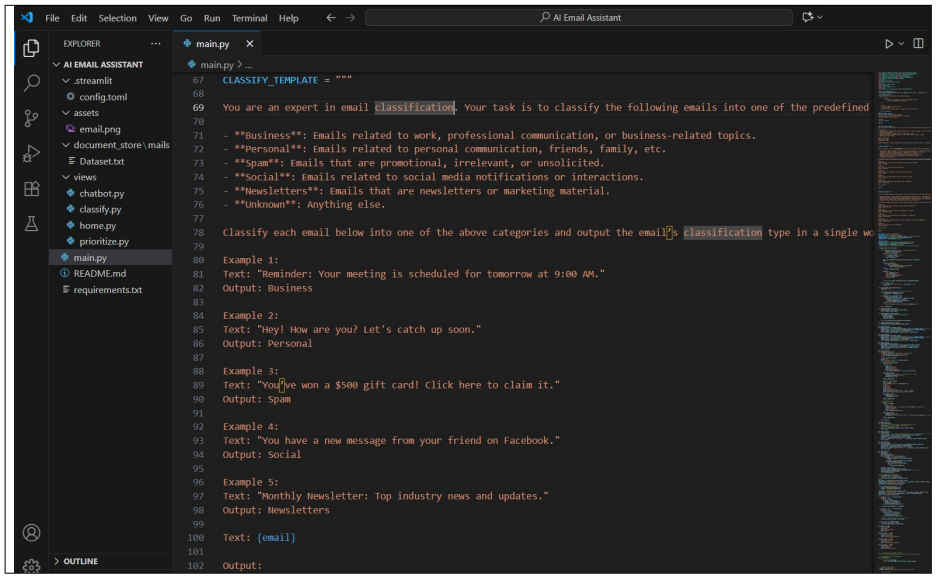


Figure 5.17: main.py component 4

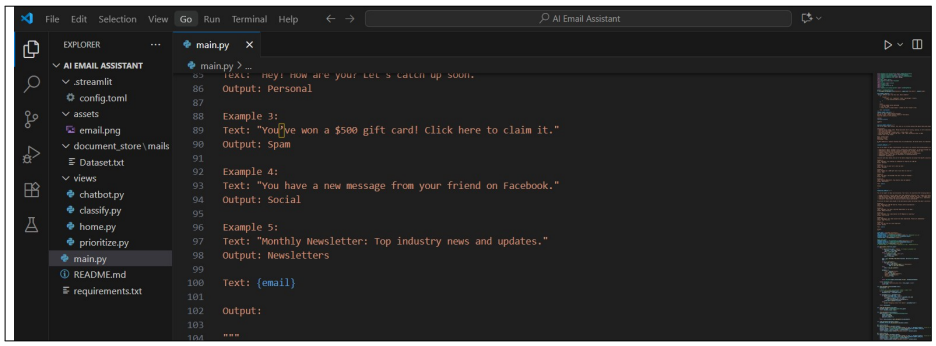


Figure 5.18: main.py component 5

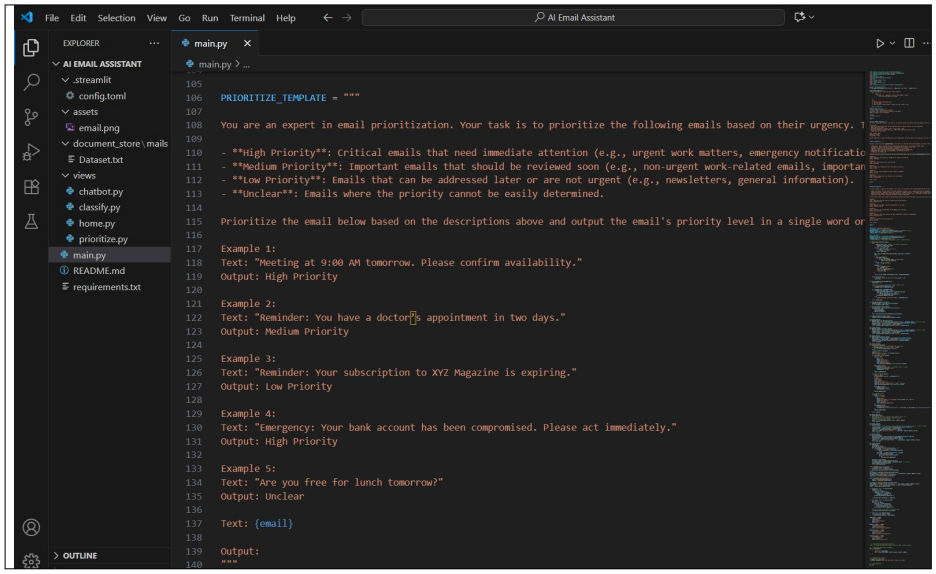


Figure 5.19: main.py component 6

Functions part

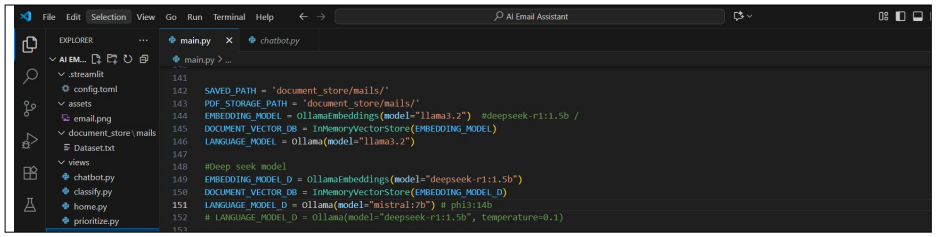


Figure 5.20: main.py component 7

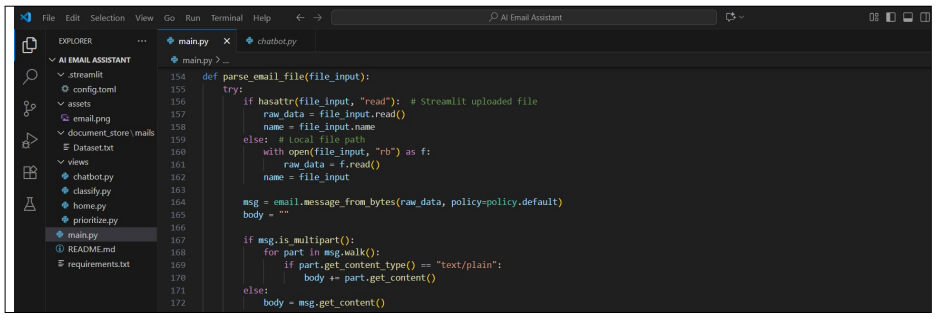


Figure 5.21: main.py component 8

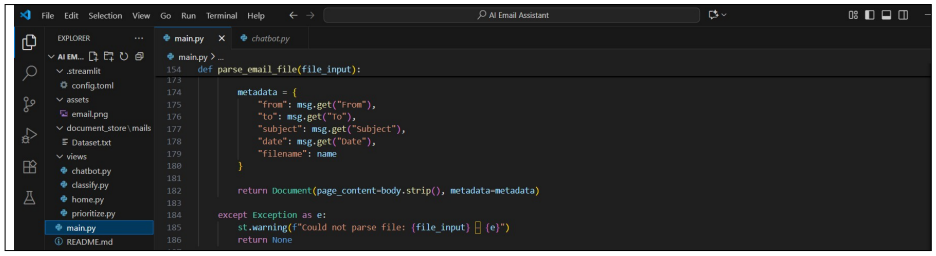


Figure 5.22: main.py component 9

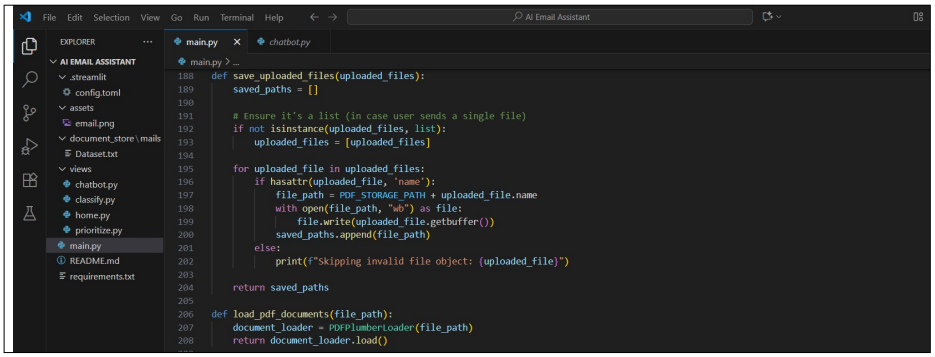


Figure 5.23: main.py component 10

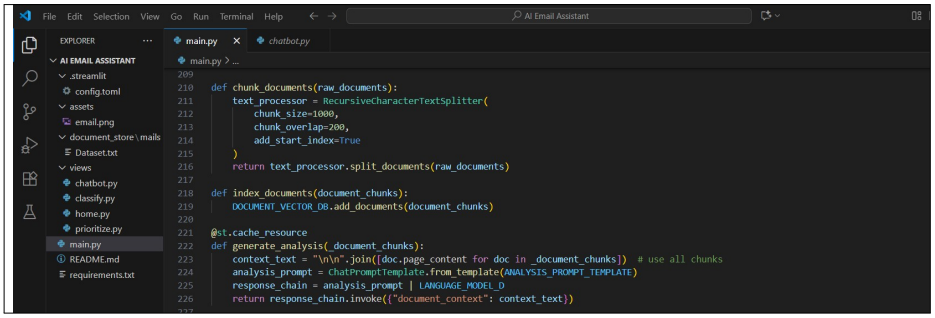


Figure 5.24: main.py component 11

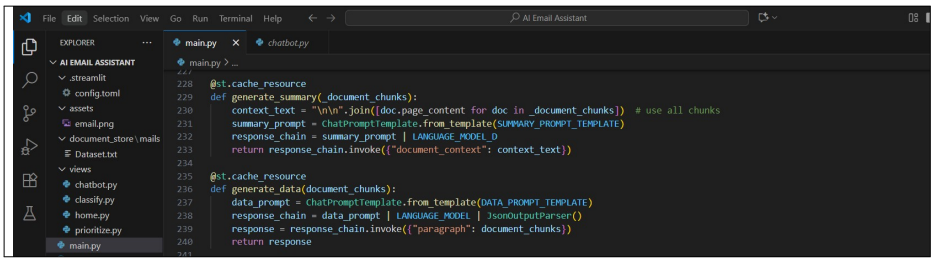


Figure 5.25: main.py component 12

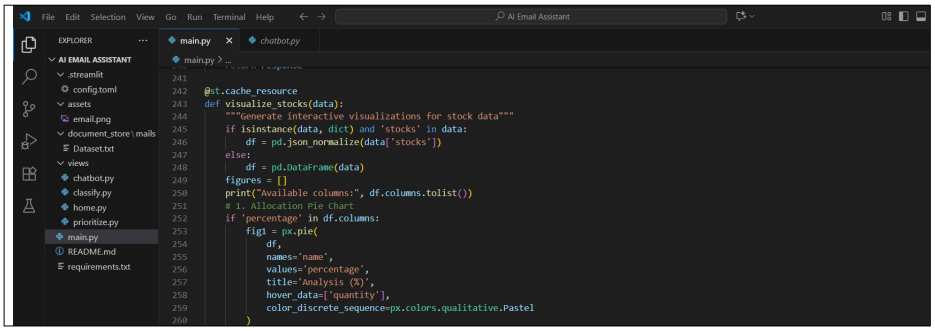


Figure 5.26: main.py component 13

Figure 5.27: main.py component 14

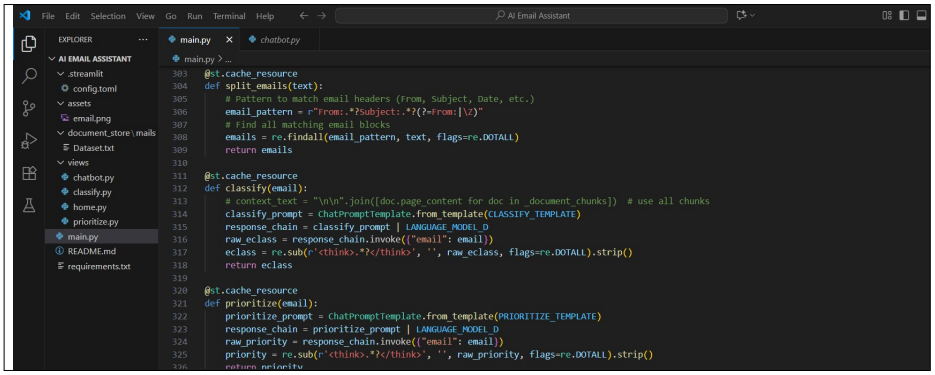


Figure 5.28: main.py component 15

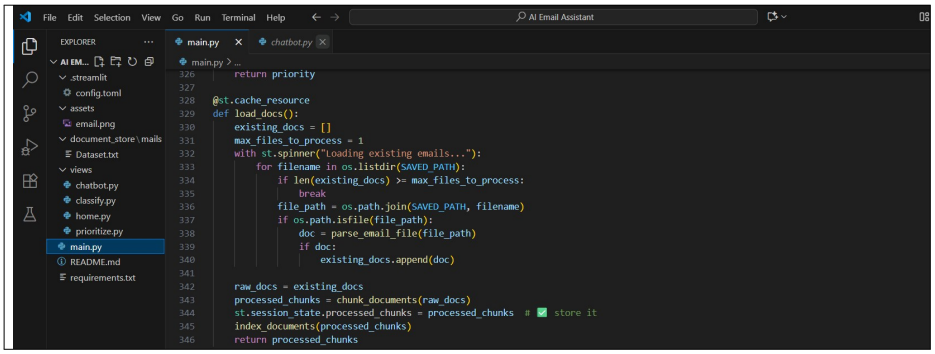


Figure 5.29: main.py component 16

Core functions are ‘load_docs()‘ that pulls in email text from storage, which receives saved emails and breaks up into chunks using ‘CHUNK_SIZE‘ and caches them in ‘st.session_state‘ sets the stage. Then ‘generate_analysis(processed_chunks)‘ pulls input for large language like mistral, deepseek to gain valuable insights. ‘generate_data(input_text)‘ reshapes it all, outputs structure. Numbers, tags, values. After that, ‘visualize_stocks(stocks)‘ takes those values, converts structured data to charts making it visible charts. These charts include person and their mail quantities, category and priority.

Then, important functions are defined such as ‘load_docs()‘ which is a loader in charge of getting email data from our storage. Then, ‘generate_analysis()‘ Later, ‘visualize_stocks()‘ converts structured data to charts. that make it catch gist of mails quickly, The‘generate_summary(processed_chunks)‘ — cut the fat, leave the meat. ‘split_emails(context_text)‘ — smash a big blob back into single mails. Then ‘classify(email)‘ and ‘prioritize(email)‘ — categories, ranks. ‘index_documents(processed_chunks)‘ boosts speed by indexing it.

The functions ‘generate_data(input)‘ generates relevant data like person, mail quantity and percentage which then passes as dataframe to ‘visualize_stocks(stocks)‘ . Raw message goes in clear, sorted, prioritized, visualized output comes out turning noisy mails to cleaner charts. The ‘st.pages‘ transits user to application views like Overview, Classification, Priority, and Chatbot. The approach taken was in a way modular, maintainable and extensible.

The Main Calling Function

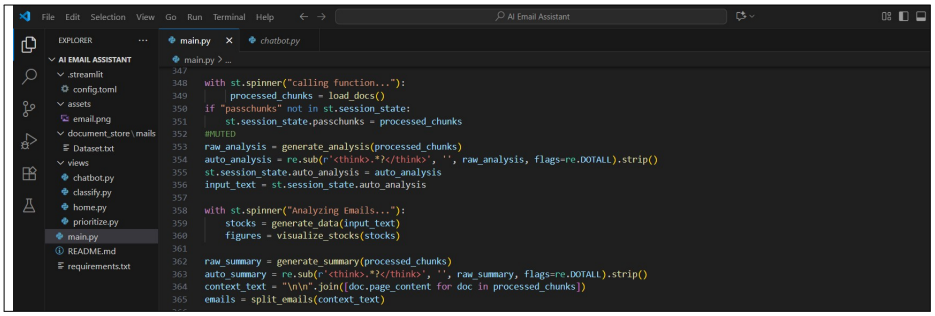


Figure 5.30: main.py component 17

This part explains how the app's core workflow works: It sets up the Streamlit page by giving it a title, an icon, and other things. It makes the navigation sidebar, which lets users choose between different perspectives. It manages the whole thing: importing the data, preparing it, sending it to our AI functions for analysis, and then eventually showing the results. The 'load_docs()' looks at path. If data exists, it pulls it in. If not, starts blank. Done. Next step: 'generate_analysis()'. Pipeline runs, text processed. Then 'generate_data()' builds single json object. Finally, 'visualize_stocks()' plots charts stores them in variable to pass to relevant pages. Shapes, colors. Data made visible.

There are helpers along the way. 'chunk_documents()'. 'parse_documents()'. They serve as suport functions to the main ones.

Overall the system unpacks loads of email, chunks it, sorts out in order where data gets structured.

Sharing the data from result Variables

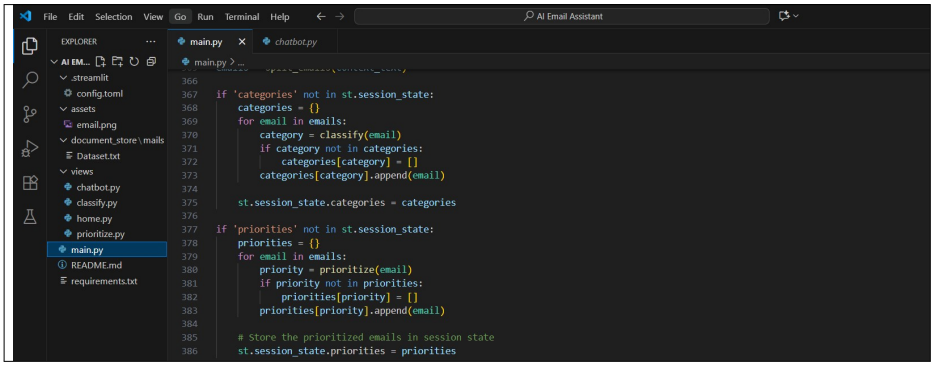


Figure 5.31: main.py component 18

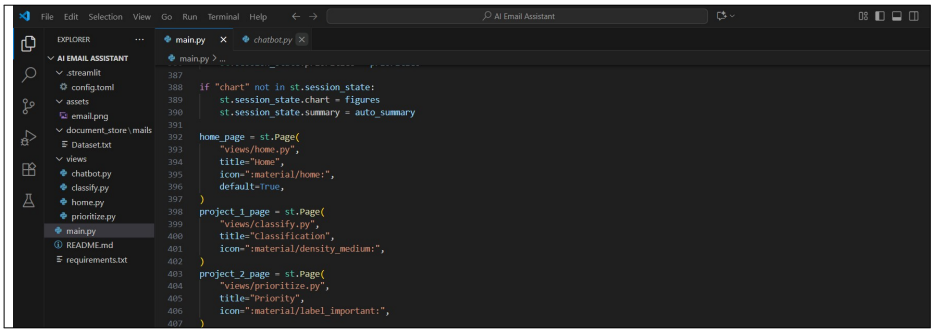


Figure 5.32: main.py component 19

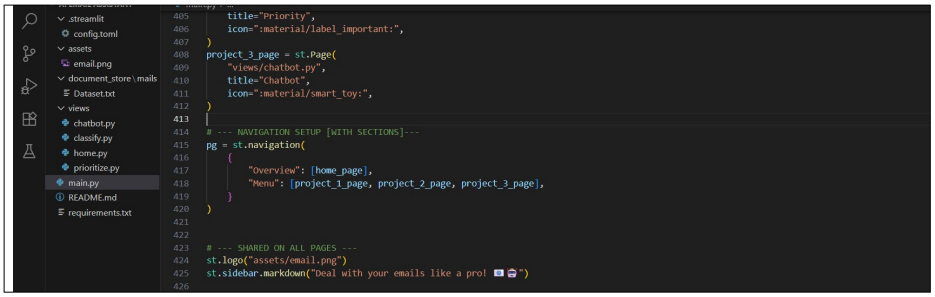


Figure 5.33: main.py component 20

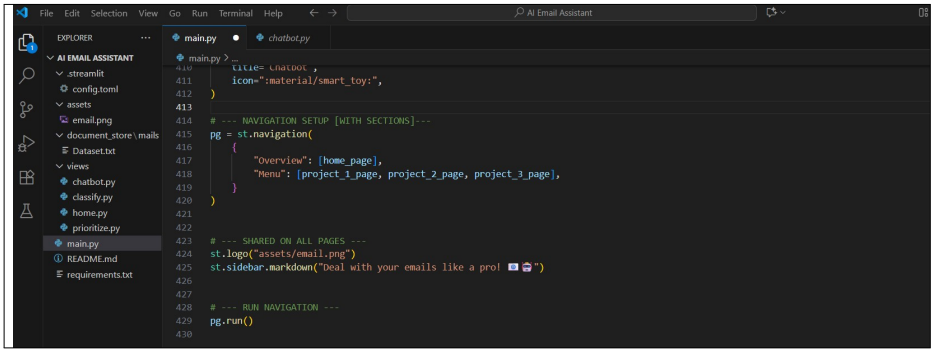


Figure 5.34: main.py component 21

We take the results of our functions, such as charts from Plotly, the sorted priorities, and the organised categories, and send them to the right sites (home.py, prioritize.py, etc.). This lets each devoted page focus just on showing its own piece of the puzzle using the information it gets from the central brain. The 'st.session_state' keeps track of variables and its data state across different views.

This has been introduced after encountering glitch on long time reloading of pages. After ‘generate_analysis()‘ executes, the results are saved in variables like ‘st.session_state.auto_analysis‘. Visualisations and summaries are preserved so that each page can access them without having to process them again.

Widgets like ”st.text_input” let users enter text and ”st.multiselect” provide them choices for variables. When we add ‘plotly.express‘, we get interactive charts. Choosing two features from a dataset and sending them to ‘px.scatter‘ generates a visualisation immediately away, which ‘st.plotly_chart‘ demonstrates. There is a steady flow of data to stop reloads.

Users can interact with widgets like ”st.text_input” takes input and ”st.multiselect” provide options for variables. When we add ‘plotly.express‘, we get interactive charts. Features from dataset passing to ‘px.pie‘ makes a visualisation right away, which ‘st.plotly_chart‘ shows. There is a consistent data transfer for preventing reloads.

In short, main.py is the director of the orchestra. It brings in the instruments (libraries), writes the sheet music (prompts), directs the musicians (functions), and makes sure that the final performance (the web app) is harmonic and powerful.

Home Component

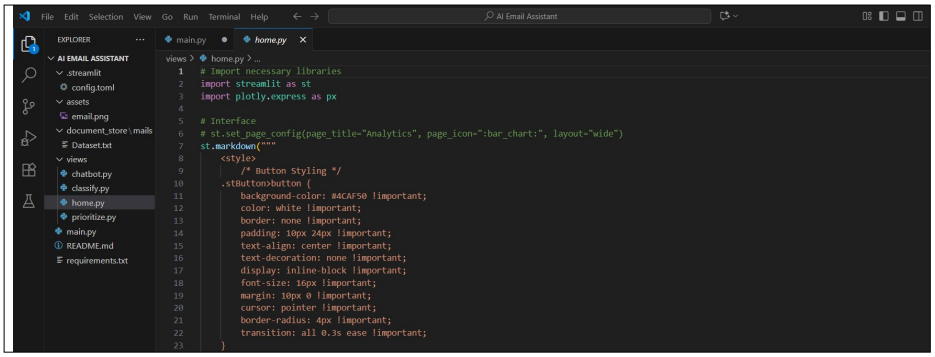


Figure 5.35: home.py component 1

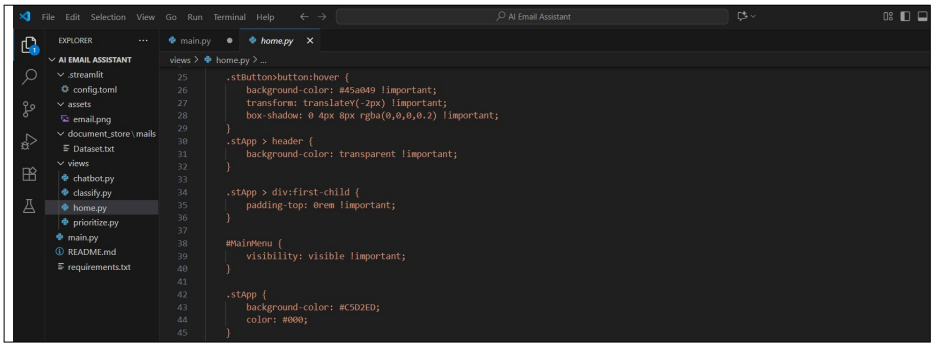


Figure 5.36: home.py component 2

The home component collects the analysis chart data and summary data from main.py and counts categories and priorities through arrays and plots three charts through namely px.pie, px.bar and px.funnel and one text part divided amoung two rows and two columns.

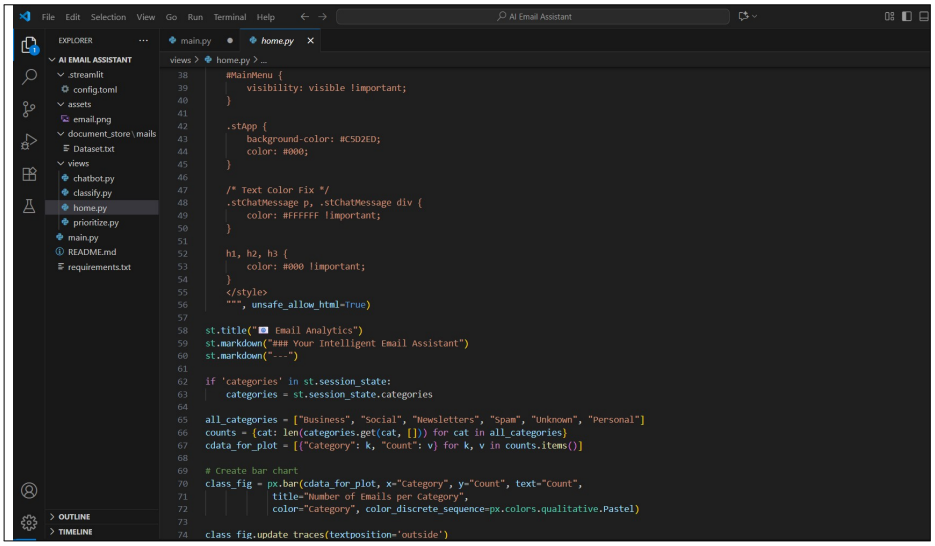


Figure 5.37: home.py component 3

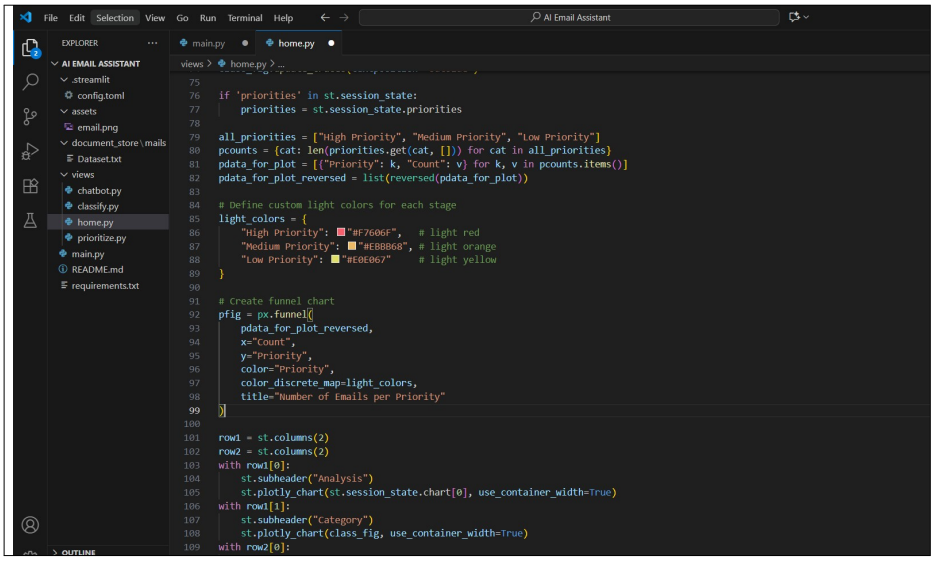


Figure 5.38: home.py component 4

```
100
101 row1 = st.columns(2)
102 row2 = st.columns(2)
103 with row1[0]:
104     st.subheader("Analysis")
105     st.plotly_chart(st.session_state.chart[0], use_container_width=True)
106 with row1[1]:
107     st.subheader("Category")
108     st.plotly_chart(class_fig, use_container_width=True)
109 with row2[0]:
110     st.subheader("Priority")
111     st.plotly_chart(pfig, use_container_width=True)
112 with row2[1]:
113     st.subheader("Summary:")
114     st.write(st.session_state.summary)
```

Figure 5.39: home.py component 5

Classification component

Initialy sets up Streamlit (import streamlit as st).Custom CSS are used with st.markdown() for styling buttons (.tab-button). Checks wether categories exist in main.py's session state (where app data is stored across interactions). categories = st.session_state.categories Creates horizontal radio buttons using st.radio() so users can pick an email category. Displays the emails of the selected category, the category title, loops through items in that category and prints them with st.markdown().

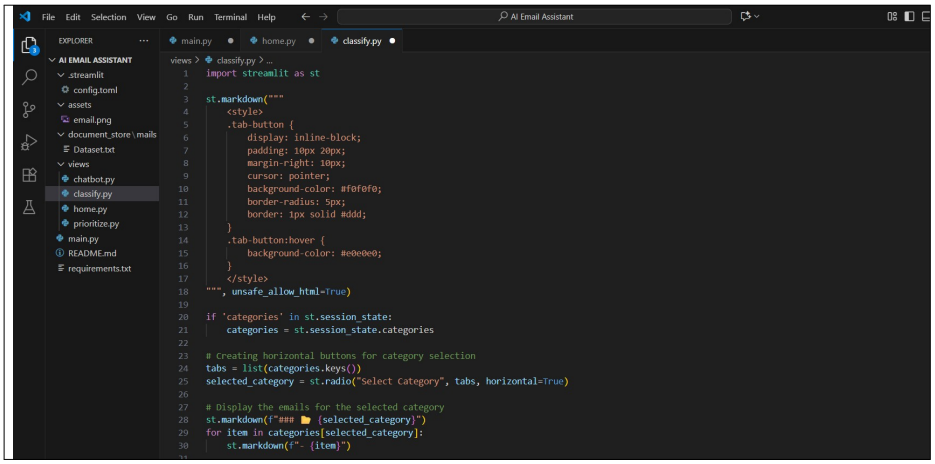


Figure 5.40: classify.py component

Prioritizing component

Checks if priorities exist in the session state. priorities = st.session_state.priorities Defines a priority order:["High Priority", "Medium Priority", "Low Priority", "Unclear"]. Displays emails grouped by priority as for each priority level, it uses expanders (st.expander) so users can expand/collapse sections. Inside each expander, it loops through the emails of that priority and shows them with st.markdown(). If there are no priorities available, it shows "No emails to display."

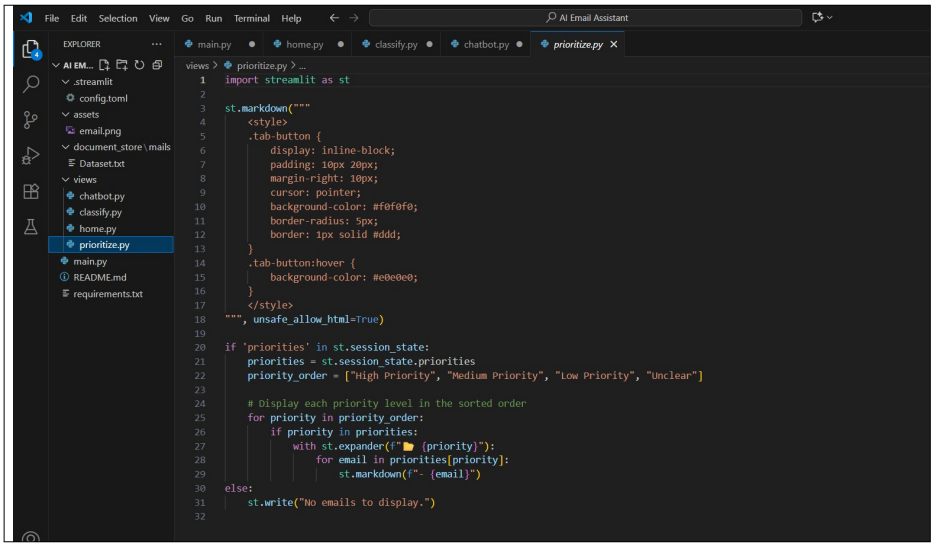
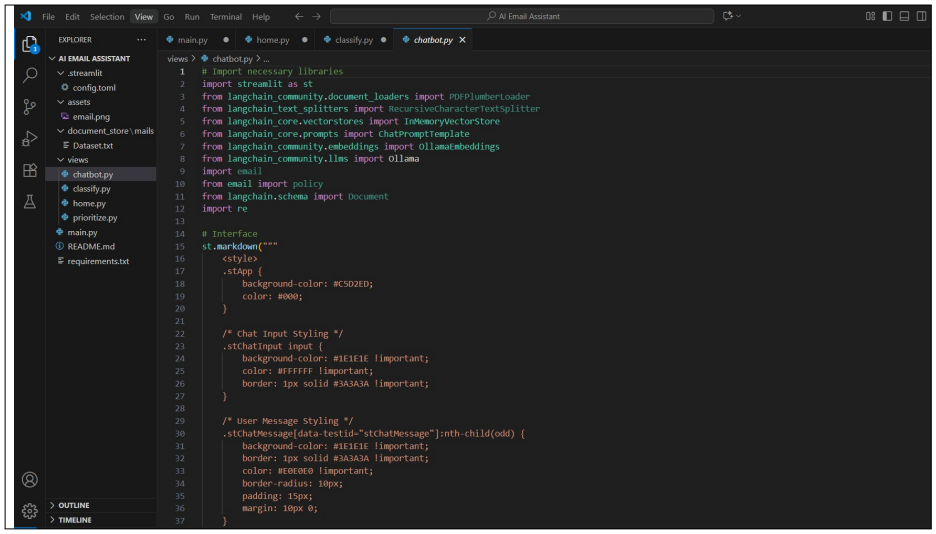


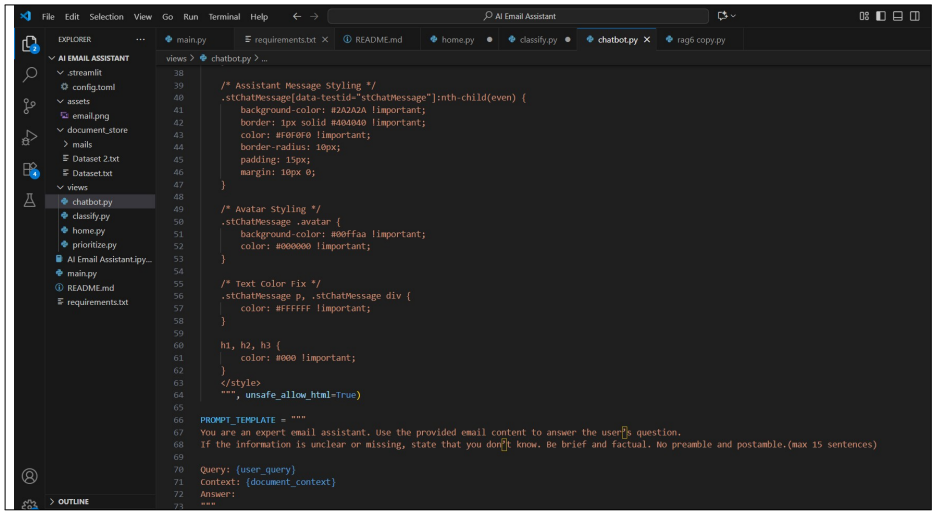
Figure 5.41: prioritize.py component

Chatbot component



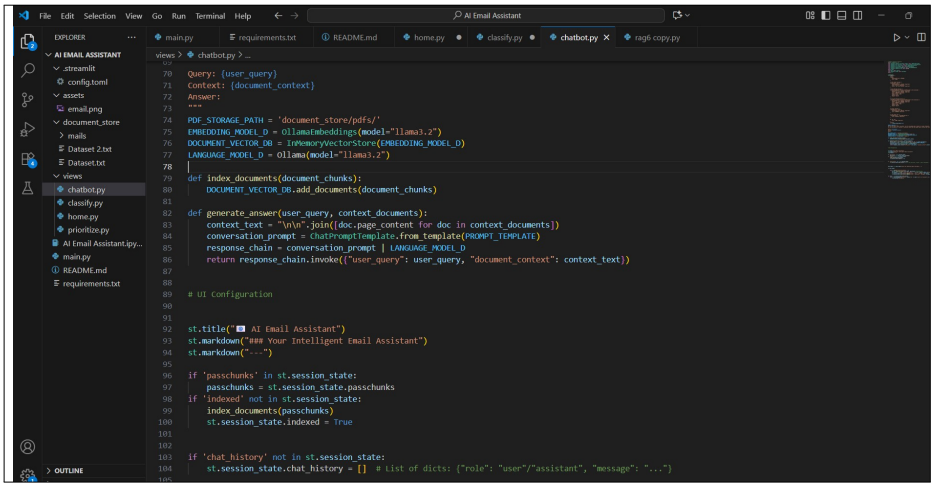
```
# Import necessary libraries
import streamlit as st
from langchain_community.document_loaders import PDFPlumberLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_core.prompts import ChatPromptTemplate
from langchain_community.embeddings import OllamaEmbeddings
from langchain_community.llms import Ollama
import email
from email import policy
from langchain.schema import Document
import re

# Interface
st.markdown("""
    <style>
    .stApp {
        background-color: #C5D2ED;
        color: #000;
    }

    /* Chat Input Styling */
    .stChatInput input {
        background-color: #1E1E1E !important;
        color: #FFFFFF !important;
        border: 1px solid #3A3A3A !important;
    }

    /* User Message Styling */
    .stChatMessage[data-testid="stChatMessage"]:nth-child(odd) {
        background-color: #1E1E1E !important;
        border: 1px solid #3A3A3A !important;
        color: #E0E0E0 !important;
        border-radius: 10px;
        padding: 15px;
        margin: 10px 0;
    }
```

Figure 5.42: chatbot.py component 1



```
    /* Assistant Message Styling */
    .stChatMessage[data-testid="stChatMessage"]:nth-child(even) {
        background-color: #2A2A2A !important;
        border: 1px solid #404040 !important;
        color: #F0F0F0 !important;
        border-radius: 10px;
        padding: 15px;
        margin: 10px 0;
    }

    /* Avatar Styling */
    .stChatMessage .avatar {
        background-color: #00ffaa !important;
        color: #000000 !important;
    }

    /* Text Color Fix */
    .stChatMessage p, .stChatMessage div {
        color: #FFFFFF !important;
    }

    h1, h2, h3 {
        color: #000 !important;
    }
    </style>
    """, unsafe_allow_html=True)

PROMPT_TEMPLATE = """
You are an expert email assistant. Use the provided email content to answer the user's question.
If the information is unclear or missing, state that you don't know. Be brief and factual. No preamble and postamble.(max 15 sentences)

Query: {user_query}
Context: {document_context}
Answer:
"""
```

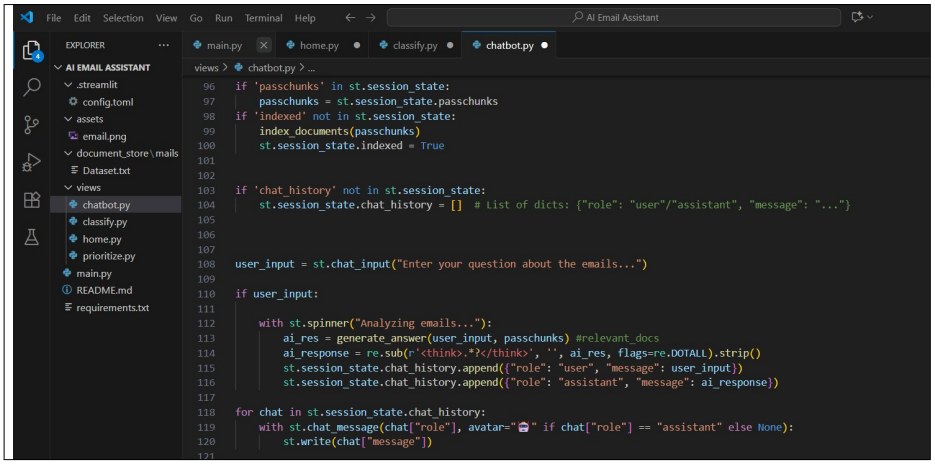
Figure 5.43: chatbot.py component 2



```
Query: {user_query}
Context: {document_context}
Answer:
"""
PDF_STORAGE_PATH = 'document_store/pdfs/'
EMBEDDING_MODEL_D = OllamaEmbeddings(model="llama3.2")
DOCUMENT_VECTOR_DB = InMemoryVectorStore(EMBEDDING_MODEL_D)
LANGUAGE_MODEL_D = Ollama(model="llama3.2")

def index_documents(document_chunks):
    DOCUMENT_VECTOR_DB.add_documents(document_chunks)

def generate_answer(user_query, context_documents):
    context_text = "\n\n".join([doc.page_content for doc in context_documents])
    conversation_prompt = ChatPromptTemplate.from_template(PROMPT_TEMPLATE)
    response_chain = conversation_prompt | LANGUAGE_MODEL_D
    return response_chain.invoke({"user_query": user_query, "document_context": context_text})


# UI Configuration


st.title("📧 AI Email Assistant")
st.markdown("### Your Intelligent Email Assistant")
st.markdown("---")

if 'passchunks' in st.session_state:
    passchunks = st.session_state.passchunks
if 'indexed' not in st.session_state:
    index_documents(passchunks)
    st.session_state.indexed = True


if 'chat_history' not in st.session_state:
    st.session_state.chat_history = []  # List of dicts: {"role": "user"/"assistant", "message": "..."}
```

Figure 5.44: chatbot.py component 3

The chat bot part consists of ‘RecursiveCharacterTextSplitter‘ processing docs, ‘OllamaEmbeddings‘ and ‘InMemoryVectorStore‘ for indexing ‘document_chunks‘ via ‘add_documents‘. The ‘generate_answer‘ function use ‘ChatPromptTemplate‘ with a ‘PROMPT_TEMPLATE‘ to format ‘user_query‘ and ‘context_text‘ for ‘ollama‘ ‘LANGUAGE_MODEL_D‘, to invoke response. The ‘st.session_state‘ holds caches session dataa for‘passchunks‘, ‘indexed‘ status, and ‘chat_history‘.



```
if 'passchunks' in st.session_state:
    passchunks = st.session_state.passchunks
if 'indexed' not in st.session_state:
    index_documents(passchunks)
    st.session_state.indexed = True


if 'chat_history' not in st.session_state:
    st.session_state.chat_history = []  # List of dicts: {"role": "user"/"assistant", "message": "..."}



user_input = st.chat_input("Enter your question about the emails...")

if user_input:

    with st.spinner("Analyzing emails..."):
        ai_res = generate_answer(user_input, passchunks) #relevant_docs
        ai_response = re.sub(r'<think>.*?</think>', '', ai_res, flags=re.DOTALL).strip()
        st.session_state.chat_history.append({"role": "user", "message": user_input})
        st.session_state.chat_history.append({"role": "assistant", "message": ai_response})

for chat in st.session_state.chat_history:
    with st.chat_message(chat["role"], avatar="🤖" if chat["role"] == "assistant" else None):
        st.write(chat["message"])
```

Figure 5.45: chatbot.py component 4

5.2 Testing

5.2.1 Testing code

Figure 5.46: Testing Code component 1

Figure 5.47: Testing Code component 2

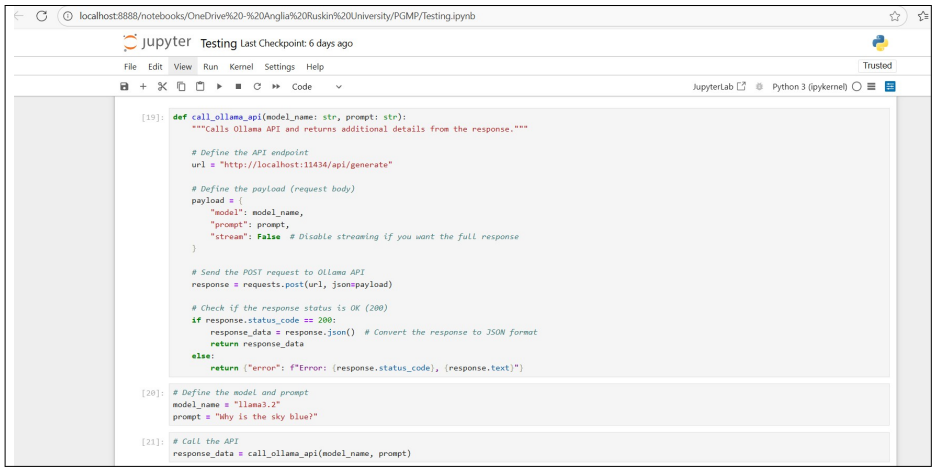
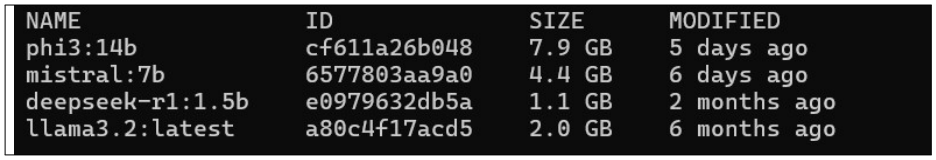


Figure 5.48: Testing Code component 3

```
response_data = call_ollama_api(model_name, prompt)

[55]: # # Print full response in a nicely formatted JSON
      # print(json.dumps(response_data, indent=4))

[22]: # Extract specific details
      response_text = response_data.get('response', 'No response text available')
      model_used = response_data.get('model', 'Unknown Model')
      created_at = response_data.get('created_at', 'Not available')
      total_duration = response_data.get('total_duration', 0)
      prompt_eval_count = response_data.get('prompt_eval_count', 0)
      eval_duration = response_data.get('eval_duration', 0)

[26]: # Print extracted details
      print(f"Model Used: {model_used}")
      print(f"Response Created At: {created_at}")
      print(f"Total Duration (in nanoseconds): {total_duration}")
      print(f"Prompt Evaluation Count: {prompt_eval_count}")
      print(f"Evaluation Duration (in nanoseconds): {eval_duration}")
      print(f"Response: {response_text}")
```

Figure 5.49: Testing Code component 4

5.2.2 Testing Output



```
NAME                ID              SIZE      MODIFIED
phi3:14b            cf611a26b048    7.9 GB    5 days ago
mistral:7b          6577803aa9a0    4.4 GB    6 days ago
deepseek-r1:1.5b    e0979632db5a    1.1 GB    2 months ago
llama3.2:latest     a80c4f17acd5    2.0 GB    6 months ago
```

Figure 5.50: Testing Model Details

```
localhost:8888/notebooks/OneDrive%20-%20Anglia%20Ruskin%20University/PGMP/Testing.ipynb

jupyter Testing Last Checkpoint: 6 days ago

File Edit View Run Kernel Settings Help                                   Trusted

Code                                                  JupyterLab  Python 3 (ipykernel)

print(f"Response: {response_text}")

Model Used: llama3.2
Response Created At: 2025-09-15T11:37:21.3728303Z
Total Duration (in nanoseconds): 5099440700
Prompt Evaluation Count: 31
Evaluation Duration (in nanoseconds): 4054000000
Response: The sky appears blue because of a phenomenon called scattering, which occurs when sunlight interacts with the tiny molecules of gases in the Ea
rth's atmosphere.

Here's what happens:

1. **Sunlight enters the Earth's atmosphere**: When the sun shines, its light travels through space and enters our atmosphere.
2. **Light encounters atmospheric particles**: The sunlight then encounters tiny molecules of gases such as nitrogen (N2) and oxygen (O2) in the atmosphe
re.
3. **Scattering occurs**: These gas molecules scatter the light in all directions, but they scatter shorter (blue) wavelengths more than longer (red) wav
elengths.
4. **Blue light is scattered**: As a result, the blue light is dispersed throughout the atmosphere, reaching our eyes from all directions.
5. **Red light passes through**: Meanwhile, the longer wavelengths of light, such as red and orange, continue to travel in a more direct path, reaching o
ur eyes only when the sun is overhead.

This is why the sky appears blue during the daytime, especially in the direction of the sun. The color of the sky can also be affected by atmospheric con
ditions, such as pollution, dust, and water vapor, which can scatter light in different ways.

It's worth noting that the exact shade of blue we see in the sky can vary depending on factors like the time of day, atmospheric conditions, and our loca
tion on Earth.

[ ]:
```

Figure 5.51: Testing Output component 1

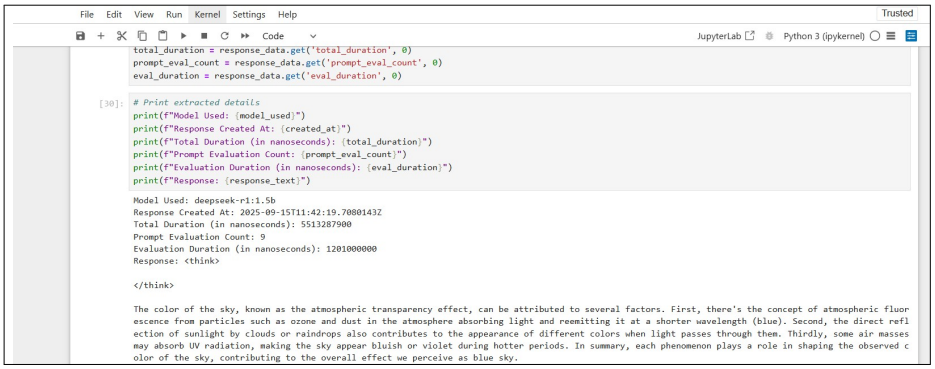


Figure 5.52: Testing Output component 2

Figure 5.53: Testing Output component 3

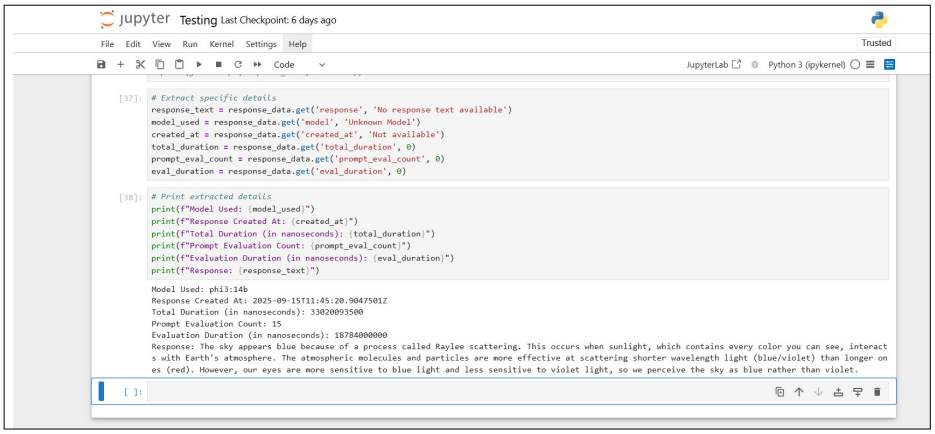


Figure 5.54: Testing Output component 4

6 Discussion and Critical Appraisal of Results

This section is a discussion on what came out of the system.A tool was built to kill boring inbox work: sort mail, rank it, cut it short, draw quick charts.

6.0.1 Results Comparison

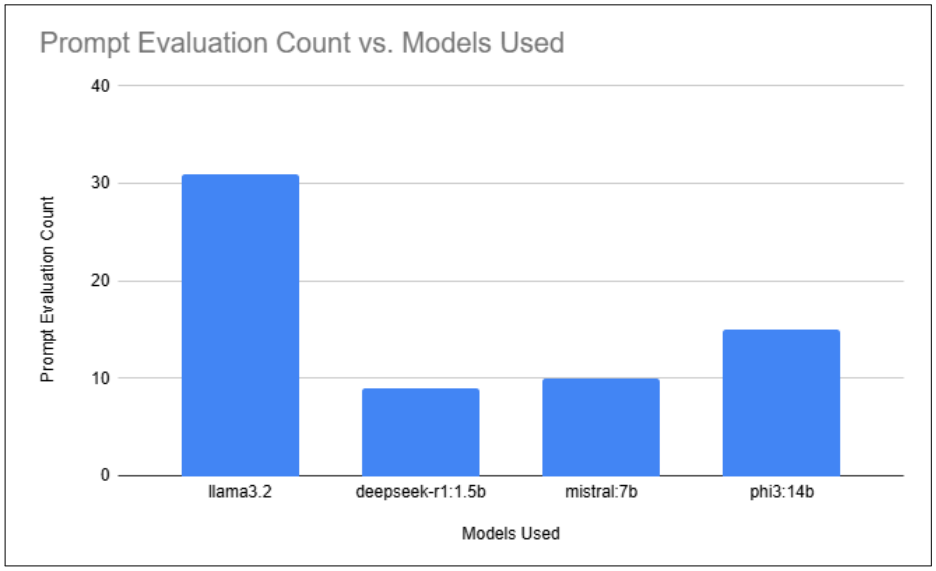


Figure 6.1: Models vs Prompt Evaluation Count

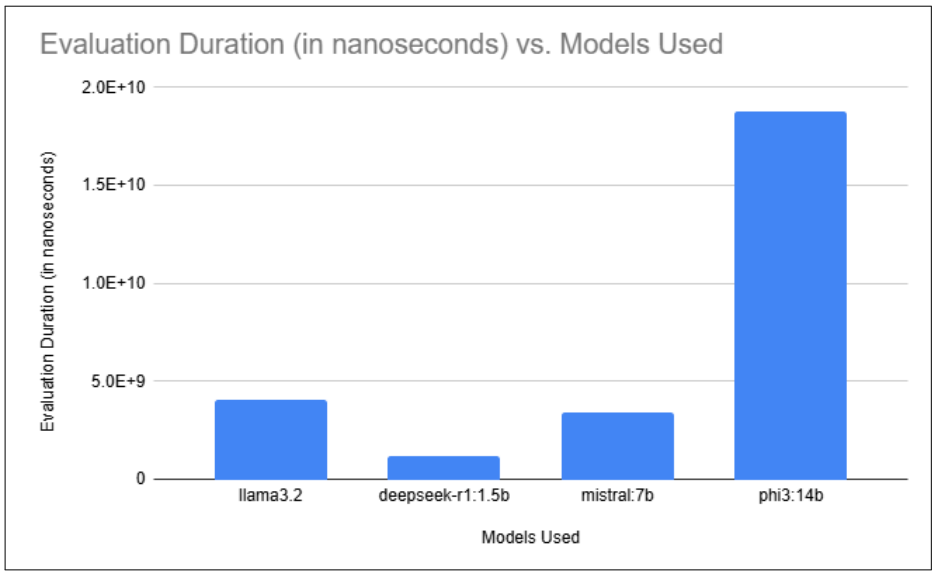


Figure 6.2: Models vs Evaluation Duration

Although Deepseek performed faster and efficient. The responses of Deepseek and phi3 were more random even after reducing the temperature. Over long run and on repetitive testing mistral was comparatively best suited for stable and timely generation of data, summaries and charts whereas llama3.2 was chosen for chatbot integration due to less errors.

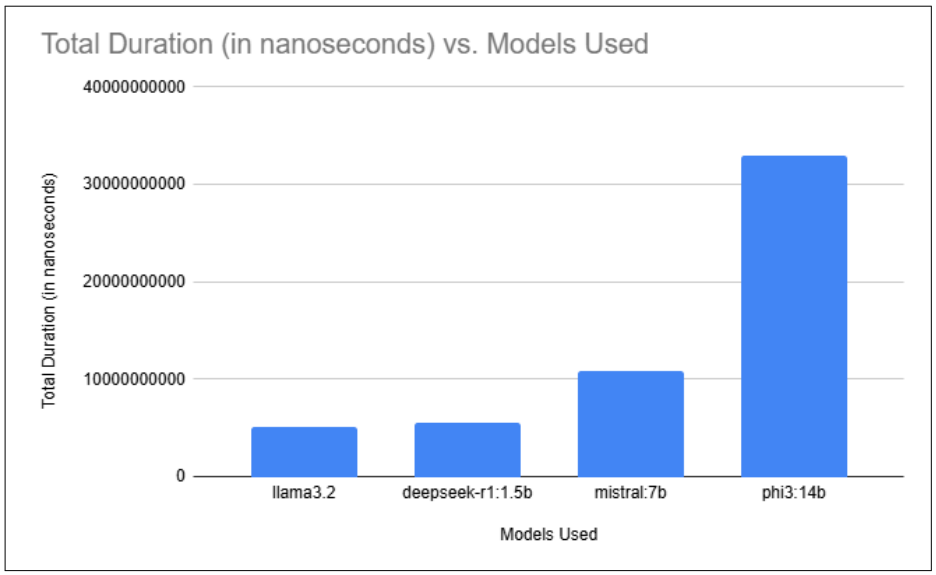


Figure 6.3: Models vs Total Duration

Tool showed strong skill classifying mail by content. Tossed messages into buckets, grouped similar messages. Handy for anyone drowning in inbox clutter. But cracks show—garbled subject lines, vague text, messy structure, wrong bucket. Priority engine another win. Pushes hot mails up, lets user tackle big messages first. Cuts clutter boosting focus. Yet not magic—ambiguous mails confuse it, some slip under radar and wrong rank. Summarizer presents short digest, fast view of inbox. Saves eyes, saves time, no need to read every single line. Works fine for most, but long, tangled mails? Some nuance lost, some detail dropped. Charts clean, easy to poke at. Spread by category, priority. Patterns pop, trends clear. Helps user spot weak spots. Still, charts live or die by upstream accuracy. Misclassified mail → bad graphs, bad story. All said, tool proves combination of LLM and charts works.

- **Strengths:** sorting, ranking, shrinking, better display.
- **Weakness:** no machine catches full human tone.

Thus, it is a solid step toward lighter inbox. Not flawless, but saves time better, lowers weight. Big deal for workers slammed with daily mail floods. Plus mental side get less inbox stress, less nagging feeling of falling behind. Summaries and auto-sorting give more grip, restore control. Cleaner headspace, more balance. Bottom line: tool hits common email pain points. Efficiency goes up, stress drops, focus returns. Future updates can lock tool as must-have, personal and work life alike. LLM engine and plain interface are not just time saver, stress saver too. Last but not least improves work-life balance.

Conclusions

Project presents a email management system. Automates key tasks. Transforms user email handling. Features include classification, prioritization, data visualization. Streamlines process for large email volumes. Large language model integration stands out as it learns from user interactions. Categorizes, prioritizes emails. Saves time meanwhile ensuring important emails to get noticed.

Data visualization part enhances system utility. Provides email data in charts that show email distribution among people and across categories. Offers insights for Informs user decisions. Certain limitations persist within the current framework. The classification model demonstrates occasional fallibility, particularly with emails employing sophisticated, nuanced, or deliberately ambiguous language. Such instances can result in mis-categorization or an incorrect priority assignment. Future development work will concentrate intensely on augmenting the model's precision and resilience by feeding more data. Functionalities may involve adding advanced features like filtering emails, integration with other tools and automated response.

Project represents significant step in email automation. Combines machine learning, visualization, user-friendly design. Powerful solution for email management. Further development holds immense potential. Improves productivity. Streamlines workflows. Innovative system revolutionizes inbox approach. Intelligently categorizing based on content and prioritizing on semantic undestanding. Helping users to find relevent mails quickly. Reduces sorting time.

Visualization features provide clear email activity overview. Identify trends, patterns. Reveal communication habit improvements. Informs task prioritization decisions.Compatibility with productivity tools opens possibilities. Reduces manual effort and boosts efficiency. Significant step in automation technology. Combination offers powerful email management. Improves productivity. Continued development promises vast benefits. Exciting prospect for businesses and individuals.

References

1. Hassabis, D. (2025) 'Google working on AI email tool that can answer in your style', The Guardian, 3 June. Available at: https://www.theguardian.com (Accessed: 15 July 2025).

2. Choe, M., Cho, H., Seo, C. and Kim, H. (2025) 'Do all autoregressive transformers remember facts the same way? A cross-architecture analysis of recall mechanisms', arXiv preprint. Available at: https://arxiv.org/abs/2025.xxxxx (Accessed: 14 August 2025).

3. D'Angelo, F., Croce, F. and Flammarion, N. (2025) 'Selective induction heads: How transformers select causal structures in context', arXiv preprint. Available at: https://arxiv.org/abs/2025.xxxxx (Accessed: 11 August 2025).

4. Ding, Y., et al. (2025) 'D² HScore: Reasoning-aware hallucination detection via semantic breadth and depth analysis in LLMs', arXiv preprint. Available at: https://arxiv.org/abs/2025.xxxxx (Accessed: 22 August 2025).

5. Khan Raiaan, M.A., et al. (2024) 'A review on large language models: Architectures, applications, taxonomies, open issues and challenges', IEEE Access, 12, pp. 12345–12367. doi: 10.1109/access.2024.3365742.

6. Kemmer, T., et al. (2025) 'Semantic partitioning through contextual attention weighting in large language models', OSF Preprints.doi: 10.31219/osf.io/ep2gq.

7. Kuang, Y., et al. (2025) 'Customizing the inductive biases of softmax attention using structured matrices', arXiv preprint. Available at: https://arxiv.org/abs/2025.xxxxx (Accessed: 8 August 2025).

8. Luo, Q., Zeng, W., Chen, M., Peng, G., Yuan, X. and Yin, Q. (2023) 'Self-attention and transformers: Driving the evolution of large language models', Proceedings of ICEICT 2023. doi: 10.1109/iceict57916.2023.10245906.

9. Matarazzo, A. and Torlone, R. (2025) 'A survey on large language models with some insights on their capabilities and limitations', arXiv preprint. doi: 10.48550/arxiv.2501.04040.

10. Shen, L., et al. (2024) 'Improving the robustness of transformer-based large language models with dynamic attention', NDSS 2024. doi:10.14722/ndss.2024.24115.

11. Shams, D., Salama, I. and Callixtus, I. (2025) 'Exploring the landscape of large and small language models: Advancements, trade-offs, and future directions', Preprints. doi: 10.20944/preprints202501.0502.v1.

12. Soydaner, D. (2022) 'Attention mechanism in neural networks: Where it comes and where it goes', Neural Computing and Applications, 34, pp. 1–20. doi: 10.1007/s00521-022-07366-3.

13. Sun, Q., Cetin, E. and Tang, Y. (2025) 'TextTransformer^2: Self-adaptive LLMs', arXiv preprint. doi: 10.48550/arxiv.2501.06252.

14. Zahmad, I., et al. (2024) 'Probabilistic multi-layer representation cascades for context-aware large language models', OSF Preprints. doi: 10.31219/osf.io/gpts3.

15. Zaki, M.Z. (2023) 'Revolutionising translation technology: A comparative study of variant transformer models – BERT, GPT and T5', Unpublished manuscript. Available at: `https://doi.org/xxxx` (Accessed: 17 August 2025).

16. Zhang, C., et al. (2024) 'GFormer: Accelerating large language models with optimized transformers on Gaudi processors', arXiv preprint. doi: 10.48550/arxiv.2412.19829.

17. Zheng, Z., et al. (2025) 'Attention heads of large language models', Patterns, 6(3), 101176. doi: 10.1016/j.patter.2025.101176.

18. Cai, D., Wang, Y., Liu, L. and Shi, S. (2022) 'Recent advances in retrieval-augmented text generation', Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval. doi: 10.1145/3477495.3532682.

19. Chen, J., Lin, H., Han, X. and Sun, L. (2023) 'Benchmarking large language models in retrieval-augmented generation', arXiv preprint. doi: 10.48550/arxiv.2309.01431 (Accessed: 9 August 2025).

20. Fan, W., Ding, Y., Ning, L., Wang, S., Li, H., Yin, D., Chua, T-S. and Li, Q. (2024) 'A survey on RAG meets LLMs: Towards retrieval-augmented large language models', ACM Transactions. doi: 10.1145/3637528.3671470 (Accessed: 17 August 2025).

21. Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Guo, Q., Wang, M. and Wang, H. (2023) 'Retrieval-augmented generation for large language models: A survey', arXiv preprint. doi: 10.48550/arxiv.2312.10997 (Accessed: 6 August 2025).

22. Jiao, D., Cai, L., Huang, J., Zhang, W., Tang, S. and Zhuang, Y. (2024) 'DuetRAG: Collaborative retrieval-augmented generation', arXiv preprint. Available at:http://arxiv.org/abs/2405.13002 (Accessed: 14 August 2025).

23. Lyu, Y., Li, Z., Niu, S., Xiong, F., Tang, B., Wang, W., Wu, H., Liu, H., Xu, T. and Chen, E. (2024) 'CRUD-RAG: A comprehensive Chinese benchmark for retrieval-augmented generation of large language models', arXiv preprint. doi: 10.48550/arxiv.2401.17043 (Accessed: 12 August 2025).

24. Quinn, D., Nouri, M., Patel, N., Salihu, J., Salemi, A., Lee, S., Zamani, H. and Alian, M. (2024) 'Accelerating retrieval-augmented generation', arXiv preprint. Available at:http://arxiv.org/abs/2412.15246 (Accessed: 25 August 2025).

25. Reichman, B. and Heck, L. (2024) 'Dense passage retrieval: Is it retrieving?', arXiv preprint. Available at:http://arxiv.org/abs/2402.11035 (Accessed: 8 August 2025).

26. Su, W., Ai, Q. and Wu, Z. (2024) 'DRAGIN: Dynamic retrieval augmented generation based on the information needs of large language models', arXiv preprint. Available at:https://arxiv.org/abs/2403.10081 (Accessed: 21 August 2025).

27. Su, W., Tang, Y., Ai, Q., Yan, J., Wang, C., Wang, H., Ye, Z., Zhou, Y. and Liu, Y. (2025a) 'Parametric retrieval-augmented generation', arXiv preprint. Available at:http://arxiv.org/abs/2501.15915 (Accessed: 10 August 2025).

28. Su, W., Ai, Q., Zhan, J., Dong, Q. and Liu, Y. (2025b) 'Dynamic and parametric retrieval-augmented generation', arXiv preprint. Available at:http://arxiv.org/abs/2506.06704 (Accessed: 18 August 2025).

29. Tian, Y., Zhang, S. and Feng, Y. (2024) 'Auto-RAG: Autonomous retrieval-augmented generation for large language models', arXiv preprint. doi: 10.48550/arxiv.2411.19443 (Accessed: 13 August 2025).

30. Xu, C., Gao, L., Miao, Y. and Zheng, X. (2025) 'Distributed retrieval-augmented generation', arXiv preprint. Available at:http://arxiv.org/abs/2505.00443 (Accessed: 16 August 2025).

31. Yan, S-Q., Gu, J-C., Zhu, Y. and Ling, Z-H. (2024) 'Corrective retrieval augmented generation', arXiv preprint. Available at:http://arxiv.org/abs/2401.15884 (Accessed: 5 August 2025).

32. Yan, S-Q. and Ling, Z-H. (2025) 'RPO: Retrieval preference optimization for robust retrieval-augmented generation', arXiv preprint. Available at: urlhttp://arxiv.org/abs/2501.13726 (Accessed: 20 August 2025).

33. Ye, F., Li, S., Zhang, Y. and Chen, L. (2024) 'R²AG: Incorporating retrieval information into retrieval augmented generation', arXiv preprint. Available at: urlhttp://arxiv.org/abs/2406.13249 (Accessed: 23 August 2025).

34. Zhang, Q., Chen, S., Bei, Y., Yuan, Z., Zhou, H., Hong, Z., Dong, J., Chen, H., Chang, Y. and Huang, J.X. (2025) 'A survey of graph retrieval-augmented generation for customized large language models', arXiv preprint. doi: 10.48550/arxiv.2501.13958 (Accessed: 27 August 2025).

35. Zou, W., Geng, R., Wang, B. and Jia, J. (2025) 'PoisonedRAG: Knowledge corruption attacks to retrieval-augmented generation of large language models', USENIX Security Symposium. Available at: `https://www.usenix.org/system/files/usenixsecurity25-zou-poisonedrag.pdf` (Accessed: 15 August 2025).

36. Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K., 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. Proceedings of NAACL-HLT 2019 (Long and Short Papers). Available at: `https://aclanthology.org/N19-1423/` (Accessed 18 July 2025).

37. Guo, T., Zhang, Y., Li, H. and Xu, J. (2025) 'DeepSeek R1:1.5B – Efficient multi-turn corporate email reasoning with small LMs', arXiv preprint. Available at: urlhttps://arxiv.org/abs (Accessed: 19 August 2025).

38. Puhakka, M. (2025) Ollama: Local deployment of large language models. GitHub. Available at: urlhttps://github.com/ollama/ollama (Accessed: 7 August 2025).

39. Cubed, 2024. RAG 2.0: Finally Getting Retrieval-Augmented Generation Right? [online] Cubed. Available at: urlhttps://cubed.run/blog/rag-2-0-finally-getting-retrieval-augmented-generation-right-05a067927589 [Accessed 16 Sep. 2025].

40. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez,

41. A., Joulin, A., Grave, E. and Lample, G., 2023. LLaMA: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971. Available at: `https://arxiv.org/abs/2302.13971` (Accessed 16 Sep 2025).

42. Meta AI, 2023. Llama 2: Open foundation and fine-tuned chat models. Meta AI Research. Available at: `https://ai.meta.com/research/publications/llama-2-open-foundation-and-fine-tuned-chat-models/` (Accessed 16 Sep 2025).

43. Hugging Face (meta-llama), 2025. Llama-3.2-3B model card. Hugging Face. Available at: `https://huggingface.co/meta-llama/Llama-3.2-3B` (Accessed 16 Sep 2025).

44. Ollama, 2025. Ollama model library / docs. Available at: https://ollama.com/library/llama3 (Accessed 16 Sep 2025).

45. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S. and Kiela, D., 2020. Retrieval-Augmented Generation for knowledge-intensive NLP tasks. arXiv preprint arXiv:2005.11401. Available at: https://arxiv.org/abs/2005.11401 (Accessed 16 Sep 2025).

46. Klimt, B. and Yang, Y., 2004. The Enron corpus: A new dataset for email classification research. CEAS 2004 (Proceedings). Available at: https://www.ceas.cc/papers-2004/168.pdf (Accessed 16 Sep 2025).

47. Johnson, J., Douze, M. and Jégou, H., 2017. Billion-scale similarity search with GPUs (FAISS). arXiv preprint arXiv:1702.08734. Available at: https://arxiv.org/abs/1702.08734 (Accessed 16 Sep 2025).

48. LangChain documentation, 2025. LangChain — framework docs. Available at: https://langchain.readthedocs.io/ (Accessed 16 Sep 2025).

49. Streamlit docs, 2019–2025. Streamlit release notes and docs. Available at: https://docs.streamlit.io/ (Accessed 16 Sep 2025).

50. The Verge / Reuters / Business Insider (various authors), 2025. Reporting on DeepSeek R1 and industry reactions (news coverage). Examples: The Verge (DeepSeek R1 availability), Reuters (DeepSeek usage), Business Insider (DeepSeek business claims). (Accessed 16 Sep 2025).

51. Cohen, W.W. (2015) Enron Email Dataset. [online] Version 2015-02-02. Available at: https://www.cs.cmu.edu/~./enron/ [Accessed 19 September 2025].

52. Talha (2025) Synthetic Email Dataset. [Dataset] Generated using OpenAI's ChatGPT v5.0.

Bibliography

1. streamlit Streamlit Inc. (2023) *Streamlit Documentation*. [Online] Available at: `https://docs.streamlit.io/` (Accessed: 16 September 2025).

2. langchain LangChain AI (2023) *LangChain Documentation*. [Online] Available at: `https://python.langchain.com/` (Accessed: 16 September 2025).

3. pandas The pandas development team (2023) *pandas: powerful Python data analysis toolkit*. [Online] Available at: `https://pandas.pydata.org/` (Accessed: 16 September 2025).

4. plotly Plotly Technologies Inc. (2023) *Plotly Python Graphing Library*. [Online] Available at: `https://plotly.com/python/` (Accessed: 16 September 2025).

5. ollama Ollama (2023) *Ollama: Get up and running with large language models locally*. [Online] Available at: `https://ollama.ai/` (Accessed: 16 September 2025).

6. rag Lewis, P. *et al.* (2020) *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. arXiv:2005.11401.

7. llama32 Meta AI (2024) *Llama 3.2: The next generation of our open source large language model*. [Online] Available at: `https://ai.meta.com/blog/meta-llama-3-2/` (Accessed: 16 September 2025).

8. deepseek DeepSeek AI (2024) *DeepSeek R1: A series of efficient and powerful language models*. [Online] Available at: `https://deepseek.com/` (Accessed: 16 September 2025).

9. mistral Jiang, A. Q. *et al.* (2023) *Mistral 7B*. arXiv:2310.06825.

10. phi3 Abdin, M. *et al.* (2024) *Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Device*. arXiv:2404.14219.

Appendices

6.1 Appendix A: Model Comparison Tables

Table 6.1: Model Inference Performance Metrics

Model	Response Created	Total Duration	Prompt Evaluation	Evaluation Duration
llama3.2	2025-09-15 11:37:21.372	5,099,440,700	31	4,054,000,000
deepseek-r1:1.5b	2025-09-15 11:42:19.708	5,513,287,900	9	1,201,000,000
mistral:7b	2025-09-15 11:43:42.959	10,848,606,100	10	3,427,000,000
phi3:14b	2025-09-15 11:45:20.904	33,020,093,500	15	18,784,000,000

*All times in UTC; durations in nanoseconds

6.2 Appendix B: Example Email Summarisation Outputs

Input Email

> Dear team, Please find attached the financial report for Q2. We need to finalise the presentation slides by next Friday. Could marketing ensure that the updated graphs are incorporated? Finance, please verify the revenue numbers. Let us meet on Wednesday to review progress. Thanks, Jane

LLM-Generated Summary

> Q2 financial report attached. Tasks: - Marketing: update graphs. - Finance: verify revenue numbers. Team meeting scheduled for Wednesday. Deadline: presentation slides due next Friday.

6.3 Appendix C: Tools and Frameworks Used

In development of this email based application, this project leveraged modern technology stack focused on local AI execution. The following tools and frameworks used:

- **Ollama** - Local inference engine for running open-weight LLMs
- **Python 3.x** - Core programming language for application development
- **Streamlit** - Web framework for creating interactive data applications
- **LangChain** - Framework for LLM-powered application development
- **Pandas** - Data manipulation and analysis library
- **Plotly Express** - Interactive visualization library
- **RAG Architecture** - Retrieval-Augmented Generation pattern implementation
- **In-memory Vector Store** - Embedding storage and similarity search
- **JSON Parser** - Standard library for JSON data handling
- **Email Parser** - Standard library for email content extraction
- **Itertools** - Efficient looping and iterator building
- **RE (Regex)** - Pattern matching for text processing
- **VS Code** - Primary development environment

AI Models Utilized

The application supports multiple local LLMs through Ollama, this project utilized them in configurations:

- **Llama 3.2** - Meta's latest open-weight model (parameter size varies by variant)
- **DeepSeek R1 1.5B** - 1.5 billion parameter model by DeepSeek AI
- **Mistral 7B** - 7 billion parameter model by Mistral AI
- **Phi-3 14B** - 14 billion parameter model by Microsoft

6.4 Appendix D: Ethics Training Certificate

Figure 6.4: Ethics Training Certificate