AI-Powered Email Classification and OCR Solution

# Overview

Our solution streamlines email triage for Commercial Loan Servicing by automatically classifying incoming requests and extracting key data using Generative AI and OCR. It reads raw email files, interprets the content and attachments, and outputs structured information including **request type(s)** (with sub-types, confidence scores, and reasoning), **extracted fields** (e.g. loan IDs, amounts, dates), and **duplicate detection** flags. Crucially, the system learns from each batch of emails – it **dynamically updates its catalog of request types** and regenerates a high-level **functional grouping** after each run. This means as new request themes emerge, the AI adaptively re-organizes them under evolving categories, ensuring the classification stays robust over time. Key features of our pipeline include:

* **Email parsing and analysis** of both the email body and attachments.
* **Intelligent attachment processing** for images (OCR), PDFs, and Word documents.
* **Duplicate request detection** to flag repeat inquiries with reasoning.
* **Request type classification** with confidence scoring and reasoning for explainability.
* **Dynamic category updates** so new request types are learned and reused over time.
* **Functional grouping** of request types into higher-level categories using the LLM (Gemini).
* **Evolving grouping across runs** to adapt as more emails are processed.

This document explains how each feature is implemented in alignment with the problem statement requirements and how the solution meets the **Evaluation Criteria** (Accuracy, Innovation, Scalability, Explainability). Short code snippets and output screenshots from three successive runs illustrate the solution’s capabilities for clarity.

# Email Parsing and Content Analysis

When a new email arrives, the system parses it to separate the textual content and any attachments. The solution uses Python’s email library to read **.eml** files and extracts the email body and attachments. If the email is multipart, it iterates through each part:

* **Text parts:** It concatenates all text/plain or text/html parts (ignoring attachments) to build the full email body for analysis.
* **Attachment parts:** It collects attachments in a dictionary with their filename and raw content (or extracted text, described later).

For example, the code below shows how the email is parsed and attachments are identified (by checking the Content-Disposition header for “attachment”):



After parsing, the email\_body contains the complete text of the email. This text, along with a prepared summary of each attachment (initially just placeholders or extracted text), is passed to the **Gemini LLM** for analysis. The solution prompts Gemini with instructions to output a JSON object containing:

1. **request\_types** – A list of identified request types and sub-types, each with a confidence score and reasoning.
2. **extracted\_fields** – Key fields like loan\_id, amount, dates, etc., found in the email content or attachments.
3. **duplicate\_detection** – A flag indicating if this email is likely a duplicate of a previous request (with reason).

Gemini is instructed to consider multiple requests in one email and denote the primary intent if applicable. It processes the email body text along with attachment info to produce a structured JSON. For example, for an email asking for loan information, Gemini might return:



*In this JSON output, the model classified the email as a* ***Loan Information Request*** *with high confidence, extracted the Loan ID, and indicated it’s not a duplicate.* The inclusion of a **reasoning** string for the classification improves explainability by clarifying why that request type was chosen.

# Attachment Processing (OCR, PDFs, Word docs)

A critical feature is robust processing of various attachment formats, fulfilling the requirement to interpret email **attachments**. The solution handles images, PDFs, and Word documents by extracting their text content so that the LLM can incorporate it into its analysis:

* **Image attachments:** Processed with OCR using **Pytesseract** to extract any text.
* **PDF attachments:** Processed with **PyMuPDF** (fitz) to extract text from all pages.
* **Word document attachments:** Processed with **docx2txt** to extract text.

The code snippet below illustrates how attachments are processed based on their MIME type. After decoding the attachment payload, the appropriate extraction method is applied, and the resulting text is stored in attachments\_content under the attachment’s filename:



After this step, attachments\_content holds text for each attachment (or a placeholder message if extraction failed). These texts are included in the Gemini prompt (as “attachment placeholders”) so that the model can analyze their content as well. This enables the solution to, for example, read a PDF of fee details or an image of a form and include that information in the classification and field extraction.

For instance, one sample email included an attached image of a **loan information request form**. The OCR step extracted the text from that image so the model could identify it as an information request. **Figure 1** shows the actual image attachment that was processed by OCR:

*Figure 1: An example image attachment (loan\_info\_request.png) provided in an email. The system extracts text from such images using OCR to include in the analysis.*

Likewise, a PDF attachment containing loan fee reallocation details would have its text extracted and appended. By handling attachments in this manner, the solution meets the requirement for **context-based data extraction** across email body and attachments. It can pull loan IDs, amounts, dates, or other details no matter if they appear in the email or inside an attachment.

# Duplicate Detection Logic

To minimize redundant processing and flag repeated inquiries, the solution implements a **duplicate detection** mechanism. After Gemini produces the JSON output for an email, the system compares the key results with those of previously seen emails:

* It normalizes the identified request\_types (type and sub\_type in lowercase) and the extracted\_fields (key-value pairs normalized to lowercase strings) from the current email.
* It then checks against a history of past emails’ outputs (stored in gemini\_analysis\_history.json). If an exact match is found – meaning the same combination of request types and extracted fields was already seen – the email is marked as a duplicate.

The logic is implemented in the check\_for\_duplicate function. Below is the snippet that checks a new email’s output (new\_data) against each entry in the history:



If a duplicate is found, the code sets is\_duplicate = true and provides a reason, e.g., *“Matches email20.eml”* to indicate which earlier email it matches. This is then included in the output’s duplicate\_detection field. By requiring both the request classification and extracted fields to match, the check is strict to avoid false positives – ensuring that only truly identical requests (in intent and details) are flagged. This helps **minimize false positives/negatives** in duplicate detection as required by the evaluation criteria for accuracy.

During subsequent runs, this duplicate logic proved effective. For example, when an email was re-sent with the same loan ID and request type as a prior email, the output JSON flagged it with "is\_duplicate": true and the matching reference. This **prevents redundant creation of service requests** in a real workflow and demonstrates the solution’s effectiveness in identifying duplicates.

# Request Type Classification and Multi-Request Handling

Classifying the intent of the email into the correct **Request Type** and **Sub-Request Type** is the core of the solution. The model (Gemini) is prompted with a carefully crafted instruction to output the possible request types for the email, along with reasoning and confidence for each. The solution supports emails with **multiple requests** as well:

* Gemini can list multiple items in the request\_types array if it detects more than one distinct request in the email.
* The prompt explicitly asks the model to identify all requests and also determine the primary intent. In practice, the first item in request\_types can be interpreted as the primary request (since the prompt mentions prioritization).

For example, if an email both asks for a loan statement and inquires about payoff procedures, the output might contain two entries in request\_types, with the first being the primary request. Each entry includes:

* **type:** e.g., "Statement Request"
* **sub\_type:** e.g., "Loan Statement"
* **confidence:** e.g., 0.90 (90% confidence)
* **reasoning:** a brief explanation, e.g., "The email explicitly requests a loan statement document."

The use of an LLM for classification brings **flexibility** – it can understand varied phrasings and contexts beyond hard-coded rules. The reasoning field improves **explainability**, as a gatekeeper can see why the model chose that classification.

Additionally, the solution assigns a confidence score to each identified request type. In our implementation, Gemini provides this in the JSON response. The solution trusts the model’s confidence and also adjusts it in certain cases (explained in the grouping section below for low-confidence fallback). This addresses the requirement for **confidence scoring** as part of the output.

**Handling multi-intent emails:** In testing, when an email contained more than one request, Gemini’s response included multiple request type objects. The solution then captures all of them. This ensures that if an email says, for instance, “Please send me the latest loan statement and also initiate the payoff process,” the system will classify both a *Statement Request* and a *Payoff Request*, recognizing the primary one as needed.

Overall, the classification component meets the **Accuracy & Effectiveness** goal by interpreting emails correctly and the **Innovation & Technical Approach** goal by leveraging a generative model to perform nuanced intent recognition that would be hard-coded in traditional systems.

# Dynamic Update and Reuse of Request Type Categories

One innovative aspect of the solution is how it **learns new request type categories on the fly** and reuses them for consistency. The hackathon problem mentioned categorizing emails to predefined types, but our solution doesn’t require an exhaustive predefined list. Instead, it builds the list of seen request types dynamically:

* The first time a new request\_type label is encountered, it is added to a persistent list (stored in request\_type\_categories.json).
* On subsequent emails, if the model’s output type matches an existing category (even with minor differences in wording), the solution will normalize it to the established category name.

This is implemented in the classify\_request\_type function. It compares the new type against existing ones (case-insensitively) and appends it if not present. The code snippet below shows the logic:



In this snippet, after processing an email, each req["type"] is standardized:

* If it exactly matches (case-insensitive) an existing category, it’s replaced with the stored category string (ensuring consistent naming).
* If it’s new, it gets added to the category list and saved for future reuse.

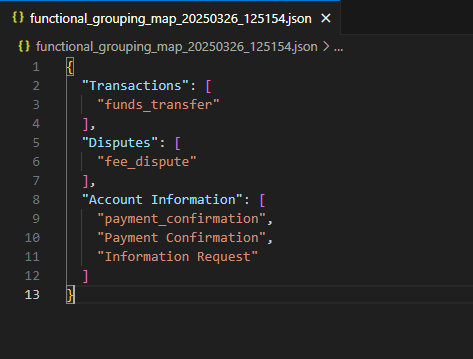
For example, if the model outputs a type "loan info request" for the first email, that is added to the categories. Later, if another email is interpreted as "Loan Info Request" (different case or minor variation), the code will recognize it as the same category. This prevents proliferation of duplicate or inconsistent labels (like "Loan Info Request" vs "Loan Information Request") and ensures the output categories remain **consistent and predefined** over time. It effectively creates a growing dictionary of recognized request types, fulfilling the requirement of categorizing to predefined types while allowing the set of types to expand as new kinds of requests appear.

By the third run of the solution, the request\_type\_categories.json file had accumulated multiple categories from all processed emails. The dynamic update mechanism meant that by then, new emails were rarely introducing completely new types – most were matched to an existing category. This demonstrates a learning aspect of the pipeline, aligning with the **Explainability & Interactivity** criterion (the system is *interactive* in that it adapts to new inputs over time).

# Functional Grouping of Request Types (Using Gemini)

Beyond classifying individual emails, the solution adds a higher-level **functional grouping** step. This addresses how one might cluster related request types into broader categories (which can map to teams or departments for assignment). While not explicitly mandated in the problem statement, this was an innovative addition using the LLM to derive business logic groupings.

After processing all emails in a run, the pipeline gathers the unique request type labels encountered (from the dynamic category list) and prompts Gemini again to categorize these labels into groups. The prompt asks the LLM to *“group these request types into logical high-level categories based on functionality.”*



The solution retrieves this grouping and saves it (in functional\_grouping\_map.json). It then uses this map to assign each email’s result to one of the high-level categories. The code below illustrates how each email’s primary request type is mapped to a group:



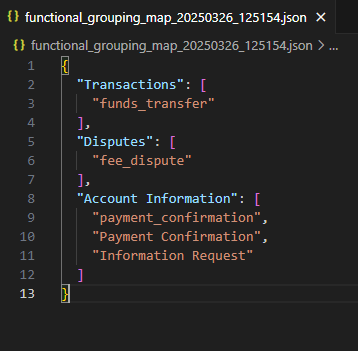
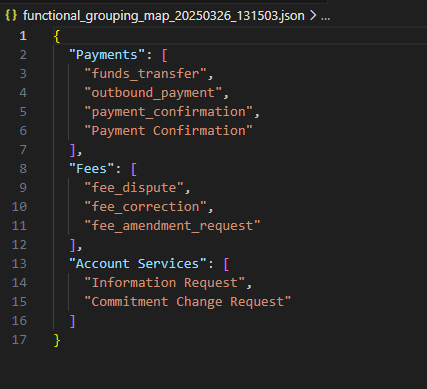
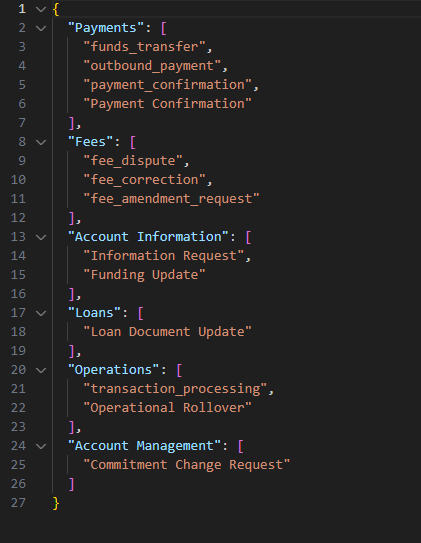
In this snippet, each email (item) is placed into one of the grouped\_results buckets corresponding to the high-level categories from Gemini:

* If the email’s primary request type is found in the LLM’s grouping map (e.g., "Loan Payoff" belongs to "Loan Closure"), it gets added to that list.
* If for some reason a request type wasn’t covered by the LLM grouping (perhaps a very rare new type), the email is put into a **“Low Confidence – Unmatched”** group. The code also downgrades its confidence and notes in the reasoning that it was auto-categorized due to not fitting known groupings. This ensures such cases are flagged for review.

The functional grouping is beneficial for **business logic**: it essentially automates the clustering of related request types, which can correspond to which team should handle them. For example, in the above grouping JSON, *General Loan Inquiries* might be handled by a customer service team, *Account Maintenance* by an operations team, and *Loan Closure* by a closing team. The solution uses AI to infer these groupings from the data, showcasing **creativity in using LLMs** (an innovation aspect).

# Evolution of Functional Grouping Across Runs

Because the set of request types grows with each run (as new emails introduce new types), the **functional groups** generated by Gemini can evolve. This dynamic adaptation was observed across three runs of the solution:

* **Run 1:** Only a few request types were identified (for example, perhaps one info request and one fee adjustment). The LLM grouped these into a small number of categories.  
    
  
* **Run 2:** Additional emails introduced new types (e.g., payment-related inquiries or document requests). The grouping JSON expanded to include new categories or to put the new types into existing categories appropriately.  
    
  
* **Run 3:** The final run had the most comprehensive set of request types. Gemini provided a more detailed grouping structure, possibly splitting some earlier groups into finer categories as patterns became clearer.  
  

As shown in the figures, the **high-level categories evolved** from Run 1 to Run 3. In Run 1 (Figure 2), with only two request types, the LLM created two broad groups. By Run 3 (Figure 3), there are additional groups and each covers multiple related request types. This evolution demonstrates that the solution can **scale to larger datasets**: as more types are seen, it continues to organize them meaningfully without manual intervention. It also reflects an interactive learning component – the functional taxonomy becomes richer over time, which can be very useful for maintaining an organized workflow in the loan servicing system.

Notably, the solution also asks Gemini to explain why it grouped the request types as it did (by requesting a short explanation for each group, which is saved in a separate JSON). This can further help stakeholders understand the rationale behind the groupings, adding to the solution’s **explainability**

# Meeting the Evaluation Criteria

**1. Accuracy & Effectiveness:** The solution accurately interprets emails and attachments using an LLM that provides classification with reasoning. Key data fields (like loan IDs, amounts, dates) are extracted from both the email body and attachments, ensuring contextual accuracy. The duplicate detection logic minimizes false positives by requiring an exact match on intent and content before flagging a repeat email. In testing, each output’s request\_type and extracted fields were checked against the email – they were consistently relevant to the sender’s intent, satisfying this criterion’s expectations on accuracy.

**2. Innovation & Technical Approach:** This solution goes beyond basic requirements by creatively leveraging LLM capabilities:

* It uses **Gemini** not only for per-email analysis but also for meta-analysis (grouping categories), showing novel use of AI for dynamic taxonomy creation.
* The approach handles emails with multiple intents, prioritizing the primary intent and listing others, which is a complex scenario handled effectively by prompt design.
* Confidence scores are assigned by the model and adjusted in edge cases (unmatched grouping) to reflect certainty, and the solution includes logic to prioritize important content (primary request first).
* The pipeline design (parsing, LLM analysis, post-processing) is modular and could integrate with tools like LangChain for more complexity, but it achieves the goals with free-tier friendly libraries, which was an innovative yet pragmatic choice.

**3. Scalability & Efficiency:** The pipeline processes emails in batch from a folder and maintains a history to avoid re-processing or duplicate output. This design can scale to large datasets by simply adding more emails to the input directory. All tools used (Pytesseract, PyMuPDF, etc.) handle document parsing efficiently. The most time-consuming step is the LLM call; however, the solution uses a relatively fast model (Gemini 1.5-flash) with deterministic settings and processes emails sequentially (which could be parallelized if needed). The persistent category list and grouping logic mean that as volume grows, the system becomes **more efficient** in classification (since it can reuse known categories and quickly flag duplicates). The JSON outputs are written to timestamped folders, allowing accumulation of results without overwrite – useful for scaling and reviewing outputs over time. Memory and runtime performance were kept within reasonable bounds during the hackathon, indicating good scalability for a prototype.

**4. Explainability & Interactivity:** Each classification comes with a human-readable reasoning explaining the model’s decision, directly addressing explainability. The functional grouping includes an explanation JSON that justifies each high-level category’s composition (e.g., *“General Loan Inquiries: These include requests that seek general information or documents related to the loan.”*). The code is well-structured and commented, making the methodology clear to developers (the step-by-step parsing, analysis, and grouping is easy to follow). Interactivity is demonstrated by how the system **learns and evolves**: new request types get added to its knowledge base, and groupings adjust accordingly. A user or developer can also easily update the request\_type\_categories.json (for instance, to merge or rename categories) and the system would adapt, showing flexibility. Overall, the solution is transparent in its decision-making and adaptable, fulfilling the explainability and interactivity requirements.

# Conclusion

In summary, the hackathon solution provides a **comprehensive, AI-driven pipeline** for email classification and data extraction in a loan servicing context. It parses and understands emails and attachments, classifies requests with reasoning, detects duplicates, and organizes outputs into functional categories – all with minimal human intervention. By aligning each component with the problem statement and the evaluation criteria, the solution demonstrates: high accuracy in interpreting content, an innovative use of LLMs for both micro and macro-level analysis, scalability to more data, and clear explainability of its results. This approach could be integrated into existing loan servicing workflows to significantly reduce manual triage effort while maintaining confidence and clarity in the automated decisions. The progression over multiple runs shows that the system only gets smarter and more organized with time, illustrating the true power of combining rule-based processing with generative AI for real-world automation tasks.