ChatGPT

Gmail Hub: Financial Email Intelligence Service – System Specification & Implementation Guide

Ingestion

Gmail API Push vs. Polling

For real-time email ingestion, **Google's push notifications via Gmail API (Cloud Pub/Sub)** are recommended over periodic polling. Push notifications eliminate the constant polling overhead by notifying our service whenever new emails arrive or mailbox changes occur [1] [2]. This improves performance and timeliness, as our backend gets near-instant alerts instead of repeatedly checking Gmail. In a push setup, we register a **Pub/Sub topic** and call Gmail's `watch` API on the user's mailbox (filtered to relevant labels, e.g. Inbox) [3] [4]. Gmail will then send a Pub/Sub message for each change, containing a `historyId` (but not the email content) [5]. Our service must follow up with Gmail API calls (e.g. `messages.get` or `history.list`) to retrieve the full email data when notified [6].

Push has clear benefits in scalability (avoiding needless API calls) and speed, but it requires **initial setup and maintenance**. We must **renew the watch** at least every 7 days (Google recommends daily) to keep receiving notifications [7]. Additionally, Gmail imposes a max notification rate of 1 event per second per user – any faster changes might be dropped, though this is rare [8]. We'll implement a **fallback polling mechanism** for robustness, as Google notes that in extreme cases push messages can be delayed or dropped [9]. For example, if no notification has been received in a certain interval (e.g. 20 minutes), the system can proactively poll Gmail via the history API or a direct messages list to catch up [9]. This aligns with best practices to ensure the application still syncs even if push fails. (Indeed, others have encountered rare Gmail push glitches and used temporary polling as a stopgap [10].)

Alternatively, a simpler design is to use **polling** only – periodically querying Gmail (e.g. every X minutes) for new messages. Polling is easier to implement (no Pub/Sub setup) and may suffice for low-volume or less time-sensitive use. However, aggressive polling can hit rate limits and wastes resources when no emails are new [1]. Polling introduces latency proportional to the interval, whereas push is immediate. Therefore, **our recommendation** is to use **push notifications** for primary ingestion, with a periodic light polling as a safety net and for certain operations like initial backfill. This hybrid approach optimizes performance while covering edge cases of missed events.

Storing Emails in `email_ingestion` Table

All incoming Gmail messages (initial historical ones and new ones) are stored in a persistent database table called `email_ingestion`. Each record in this table contains the raw email content and key metadata. We store the **raw MIME body as a binary blob** (or Base64-encoded) to preserve fidelity [11]. Emails can contain various encodings and even binary parts; treating the raw message as opaque bytes ensures nothing is lost

or corrupted by incorrect text encoding assumptions [11]. In a PostgreSQL context, for example, a `BYTEA` column is suitable for the raw email bytes. Alongside the raw blob, we record metadata fields such as:

- **Gmail IDs**: the Gmail `messageId` and `threadId` (for de-duplication and linking back to Gmail if needed).
- **User ID**: internal reference to which user/account this email belongs.
- **Timestamp**: the email's sent/received date.
- **Sender/Recipients**: email addresses for from/to (useful for parsing logic and audit).
- **Subject line**: often useful for quick filtering (e.g. contains "Receipt" or "Invoice").
- **Snippet or plain-text body**: we may store a parsed text version for quick reference, though the full raw (including HTML) is retained for detailed parsing.

All emails are stored **with minimal processing initially** – just enough parsing to extract the headers and basic fields, while the full content remains available for the parsing phase. Storing emails durably allows us to re-run parsers or fix issues later by referencing the original source. It also provides an audit trail of exactly what was received.

We handle **attachments** carefully. Attachments (PDFs, images, etc.) are typically part of the raw MIME, which we keep in the blob, but for convenience and further processing we also extract them into a separate storage. For example, the system can save each attachment file in an **object storage bucket or a separate attachments table**, with fields: attachment ID, parent `email_ingestion` ID, filename, content type, and binary data or a URL pointer to the file in cloud storage. Large attachments (like high-resolution images or multi-MB PDFs) need not live as large blobs in the main database if we use cloud storage (AWS S3, GCP Cloud Storage, etc.) – we can store a reference URI instead [12]. For fast lookup, though, small attachments (e.g. text or small image files) could be stored as binary in a table. The key is that attachments are linked back to the email record via a foreign key, maintaining referential integrity.

All ingested data is **indexed** appropriately – e.g. index on Gmail messageId to avoid duplicates, index on user and date for retrieval. During ingestion, we also mark each message with an **ingestion status** (e.g. "pending parse" initially). This status will be updated as the pipeline progresses (parsed, linked, etc.).

Handling Attachments (Invoices, PDFs, Images)

Attachments often contain critical financial data (detailed invoices, receipts, photos of receipts) that must be parsed or at least stored for reference. The ingestion service will detect attachments in incoming emails via the email's MIME structure. For each attachment, the system will:

- **Save the attachment** to persistent storage (as described above).
- **Determine the type** of attachment and invoke appropriate processing:
- **PDF or document attachments**: If the PDF is text-based (contains extractable text), we will extract the text content using a PDF parser library (e.g. PDFBox or PyPDF) to feed into our parsing pipeline. If the PDF is an image scan (or a photo in PDF), then it is essentially treated like an image.
- **Image attachments** (JPEG, PNG, etc.): These could be photos of paper receipts or screenshots. For such cases, we integrate an **OCR (Optical Character Recognition)** step. Using an OCR engine (like Tesseract or a cloud OCR service), the system converts the image into machine-readable text [13]. This text is then treated as part of the email's content to be parsed. We maintain the original image for reference and possibly to show the user in the manual review UI.

- **Other formats** (Excel, Word): These are less common for receipts but if present (e.g. an Excel expense report), we might either handle separately or require manual steps. Initially, focus is on PDFs and images which are most typical for invoices and receipts.

The parsed text from attachments is **appended to or associated with the email's text content** for parsing. We might treat the email body and attachment text as one combined stream of content for the parser, or parse them separately but merge results. We also tag parsed entities as coming from attachment vs. body if needed (for traceability).

All attachments remain available via the `email_ingestion` record (or a related table) so that the final structured data can include a reference to the file. This is crucial for downstream features: for instance, a user might click on a transaction in their ledger and open the original invoice PDF to verify details. Our storage design thus ensures we can fetch the attachment easily when needed (e.g. storing the file path/URL or an ID to retrieve from blob storage).

Onboarding Phase – Bulk Historical Ingestion

When a user first connects their Gmail, they typically want to ingest **years of historical financial emails** (e.g. past receipts, bills, and invoices) to build their financial picture. This onboarding phase can involve thousands of emails. We need a strategy to ingest this data efficiently without overloading the system or hitting Gmail API quotas.

Approach: We perform an **initial bulk sync** of relevant emails. Using the Gmail API, we can query messages – for example, list all messages in the inbox (or all messages with certain labels like "Finance" or keywords). Gmail's API allows searching by query strings, so we might narrow the sync to likely financial emails by queries such as `category:finance OR subject:(receipt OR invoice OR order OR payment)` to reduce noise. The system can also leverage known patterns: as Tailride notes, an inbox scanning tool looks for tell-tale signs like the words "invoice," "receipt," "payment due," or known vendor addresses to identify relevant emails [14]. We can incorporate similar filters to prioritize financial emails first. However, to be safe, Gmail Hub will eventually ingest **all emails** (or at least all inbox emails) for completeness, but it can prioritize and batch the ingestion.

Batching: We retrieve messages in batches using Gmail's pagination (the `messages.list` API returns message IDs which we then fetch in chunks). We must respect Gmail's **rate limits** (e.g. 250 quota units/user/second and daily quotas [15]). To ingest thousands of emails, the system will throttle itself, using **exponential backoff** on API calls if needed [16], and possibly spread the sync over multiple hours/days. Since Gmail's `watch` gives us a starting `historyId`, one approach is: - **Initial full sync:** list and fetch messages (most likely using Gmail API's **sync** approach: if the historyId from `watch` is far in the future relative to the mailbox, Gmail may return a 404 requiring a full sync [17]). In that case, we perform a full sync by listing messages. - **Partial sync via history:** If the user had recently used another client and we have a starting historyId, we could use `history.list` to get recent changes. But for onboarding, assume we need to fetch all historical data.

During this bulk ingest, we mark the emails as "historical" vs "new" so that the system can treat them appropriately (for example, the user might not want to be *notified* about each old receipt added). We also likely run parsing on these in batches offline rather than in real-time user flow.

Parallelization: We can parallelize the fetching across multiple worker threads or jobs, but must be careful not to exceed per-user concurrency limits [18]. A controlled concurrency (e.g. 2-3 threads per account) can speed up the process while staying within limits.

Onboarding AI Assistance: As part of ingestion, we also employ an **AI assistant to help with onboarding** (details in a later section). For example, after connecting, the AI agent might ask the user if they want to import historical emails and might allow the user to specify how far back to go or what types of emails to focus on. It could also gather user context (e.g. "Do you have receipts outside of email to add?") and help initiate those uploads. While the emails are being ingested, the system can keep the user informed (progress indicator, etc.).

Overall, the ingestion stage ensures every relevant Gmail message and attachment is captured into the `email_ingestion` storage, setting the foundation for parsing.

Parsing

The parsing stage translates raw email content into structured financial data. It uses a hybrid approach: a **rule-based Domain-Specific Language (DSL)** for high-precision extraction from known formats, and a **Machine Learning (NER) pipeline** for handling varied/unseen formats. The combination yields accuracy with flexibility. We also implement a **confidence scoring** mechanism to route uncertain cases to manual review.

Parser DSL for High-Precision Rules

We design a simple yet powerful **Domain-Specific Language (DSL)** to write parsers for known email templates or senders. Many financial emails (receipts from Amazon, Uber ride receipts, airline ticket invoices, etc.) have consistent formats. By writing targeted rules, we can extract fields with near 100% precision for those sources.

DSL Design: The DSL will allow definition of a *parser* with: - **Trigger conditions:** e.g. matching on sender email address or specific keywords in the subject/body to identify the email type. - **Extraction rules:** typically using regular expressions or HTML selectors to capture data. The DSL can have named fields (like `amount`, `date`, `provider`, `invoiceNumber`, etc.) and assign them patterns. We might incorporate built-in patterns for common data types (currency amounts, dates in various formats). - **Transformations:** optionally, the DSL can support simple data transformations (e.g. converting a date string "Jan 5, 2025" into `YYYY-MM-DD` format, or normalizing the provider name). - **Output template:** define how the structured output is constructed from the extracted fields.

For example, a DSL definition (in pseudocode) for an **Amazon receipt** email might look like:

```
parser "AmazonOrderReceipt" {
    when senderDomain == "amazon.com" and subject ~= /Your Order/

    extract {
        orderId: regex("Order #\s*([0-9-]+)")         // e.g. Order #
123-4567890-1234567
```

```
        amount: regex("Total:\s*\$?([0-9,]+\.\d{2})")  // capture monetary
amount
        date:   regex("Order Date:\s*([A-Za-z0-9, ]+)") // capture order date
string
        items:  regexAll("Item:\s*(.+?)\s*Qty")         // capture all
occurrences of item names
    }

    normalize {
        amount: toFloat(amount)
        date: parseDate(date)
        provider: constant("Amazon")
    }

    output {
       provider = provider,
       amount = amount,
       date = date,
       referenceId = orderId
    }
}
```

In this example, the DSL checks if the sender's domain is Amazon and subject contains "Your Order". It then uses regex rules to find the order ID, total amount, date, and possibly item names. We normalize the amount to a number and date to a standard format, and output a JSON with provider, amount, date, referenceId (using orderId as a reference). This DSL would be far more concise than writing procedural code for each parse, and easier to maintain by non-developers if needed (it could even be configured via a UI by power users in the future).

Templates vs. Code: The DSL rules essentially serve as templates that the parsing engine will apply. Under the hood, the engine loads all applicable parsers, and for each incoming email, picks the first matching parser (or set of parsers) by evaluating the `when` conditions. If a parser matches, its rules run to extract data. If multiple parsers could match (e.g. a generic one and a specific one), we define a priority or specificity ordering (specific templates first).

High Precision: This rule-based approach yields very precise results for known email types. For instance, if an Uber receipt email always says "Total: $X.XX", our regex for `Total:` will reliably grab the amount. This can often be more accurate than ML for those specific cases, and it's explainable (we know exactly how the field was extracted). It's also faster (simple regex vs running a full ML model).

We will maintain a **library of parser templates** for common financial email senders (Amazon, Uber, Lyft, airlines, PayPal, credit card statements, etc.). Over time, this library can grow based on what our users receive most often. The parser registry (discussed in Ops & Governance) will version and manage these templates.

Machine Learning Pipeline for NER

While the DSL covers known formats, users may receive a wide variety of financial emails from countless providers. To handle the "long tail" of email formats, we integrate a **Machine Learning pipeline** that performs **Named Entity Recognition (NER)** and other analyses to extract key entities: **amounts, dates, provider names, references, etc.**

NER Model: We will train a custom NER model to identify entities like `B-Amount`, `B-Date`, `B-ProviderName`, `B-AccountNumber`, etc. The training data would consist of example emails and annotated spans for these entities. For example, in a sentence "You spent $45.67 at Starbucks on 2025-09-01", the model should label "$45.67" as an Amount, "Starbucks" as a ProviderName, and "2025-09-01" as a Date. We can leverage a pre-trained language model (such as a BERT or GPT-based transformer fine-tuned on financial texts) to improve accuracy with relatively fewer examples.

Our pipeline might also use additional signals: - **Regex-based entity extractors** as a first pass: For instance, we can run regex to find any number that looks like a currency (e.g. `\$?\d+(\.\d{2})`) and label those as potential Amount entities. Similarly, date regex (for multiple date formats) can find date candidates. These provide candidates that the ML model or subsequent logic can verify or refine. - **Contextual cues:** The ML model will learn context words like "total", "amount", "date", "paid by", "merchant", etc., which help identify the entity. For example, in invoice parsing, it's known that words like "Total" or "Amount Due" precede the amount [19] . The model will use such patterns. It might also learn that vendor names often appear in certain positions (like email headers or after phrases like "from" or alongside logos). We may augment training by labeling vendor names (which might align with a list of known companies, but also using context like capitalized words or known merchant list). - **Provider/Alias lists:** As we accumulate data, we can maintain a dictionary of known provider names and their variations (for example, "Apple.com" vs "Apple Store" vs "APPLE ONLINE"). The ML pipeline can incorporate this for validation (i.e. if the model thinks a word is a provider, check if it matches or is similar to an entry in our known merchant list to boost confidence).

Custom ML from scratch: We assume we are building this pipeline internally, possibly using open-source frameworks. The NER model will likely be a sequence labeling model. Each email (after being converted to plain text and possibly segmented into sentences) is run through the model to tag tokens. We then post-process the tagged entities to form structured data. For example, if it tags "Starbucks" as provider and "$5.00" as amount and "09/25/2025" as date, we collect those. If multiple amounts are found (some receipts have itemized amounts plus a total), we might use heuristic: the **largest amount** is often the total paid [19] . If multiple dates, the one near words like "Date" or "On" might be the transaction date.

The ML pipeline also can include a **document classifier** to detect the type of email (e.g. "receipt" vs "bill" vs "bank alert"). This can help; for instance, if we detect an email is a **bill/reminder** email, the logic might extract due date and amount due (different semantics than a receipt which is paid). However, to keep scope manageable, initially focusing on receipts/invoices (proof of transactions) is fine.

Integration with DSL: The parsing system will attempt rule-based parsing first if a template matches. If a DSL parser produces a confident result (all required fields captured), we might skip ML to save time (since the rule output is presumably reliable). If no template matches or the template only partially extracts data, we then run the email through the ML pipeline to extract whatever it can. We then **merge results**: for

example, maybe a template grabbed the amount but couldn't parse the date (if an email format changed), but the ML recognized the date – we then combine those.

Confidence Scoring & Manual Review Queue

For each parsed email, the system computes a **confidence score** for the extracted data. This is crucial for quality control – we only want to fully automate high-confidence parses, and send low-confidence ones for human verification in the manual review UI.

Confidence Scoring: We assign confidence in a few ways: - **Rule-based parsers** can be given a default high confidence (since they are deterministic and tested). However, even rules might have conditions – e.g., if a regex expected a currency and none was found, that parse is incomplete (confidence low). So if a DSL parser runs and finds all its targeted fields, we consider that high confidence (say 0.9 or 0.95 out of 1.0). If it only partially matches (some fields missing or unclear), confidence drops accordingly. - **ML extraction confidence:** The NER model will output probabilities for each entity (and we can also derive an overall confidence). For instance, a model like BERT can provide a score for how likely a token is part of an Amount entity, etc. We can take the **lowest confidence among required entities** as the overall confidence, or weight them. For example, if the amount and date are high confidence but the provider name is slightly lower, we might still trust the parse overall. On the other hand, if the amount's confidence is low, that's critical and should lower the overall score significantly. - **Agreement between methods:** If both the DSL and ML (or multiple ML approaches) extract the same field consistently, confidence increases. For example, if our regex finds "$45.67" and the ML also identifies $45.67 as an Amount, that reinforces confidence. If they disagree (one found an amount but another did not, or values differ), confidence is low. - **Historical data context:** Optionally, we can incorporate whether the extracted data makes sense given user's data. For example, if the provider is "Starbucks" and we have seen Starbucks transactions for this user before, that might be normal. If the provider name is something completely new or a random string, we might be less sure it was correctly extracted. However, this is an advanced nuance and not required initially.

After computing a numerical confidence (0 to 1), we apply a threshold. For instance, if confidence >= 0.8, we consider the parse **high-confidence** (auto-accepted). If between 0.5 and 0.8, it's **medium** – maybe still accept but mark for potential review. If below 0.5, definitely route to manual review. (Thresholds will be tuned based on real data and desired precision/recall balance.)

Manual Review Queue: Any parsed result that is low confidence or otherwise problematic is sent to a **manual review queue**. In practice, this means the structured data is saved with a status "Needs Review" and an entry is created for the user (or an internal reviewer) to inspect. The manual review UI will present the email (and attachment if any) side-by-side with the extracted data fields, highlighting what the system found. The user can then correct any mistakes or fill missing info and confirm. This feedback loop can later be used to improve the parsers (for example, corrections could be fed into training data or trigger adjustments in rules).

Using confidence-based routing ensures we have humans in the loop when needed, increasing overall accuracy. Microsoft's guidance on cognitive pipelines follows a similar approach: automatically process high-confidence extractions and flag low-confidence ones for human review [20]. Likewise, Google's Document AI suggests setting a confidence threshold to decide if a prediction should be manually verified [21]. We will implement this mechanism so that our users trust that critical data (like large amount transactions) are correctly captured, either by automation or by their own verification.

Example: Suppose an email from a new vendor "ABC Plumbing" comes in with amount "512.8" (missing a trailing 0 perhaps) and our ML isn't 100% sure if that's $512.80 or $51.28 due to a formatting oddity. Confidence might be low. That email would go to the review queue, where the user can see the email and confirm the amount and vendor name. The system then saves the corrected parse. Meanwhile, an Uber receipt email that perfectly matched our template with amount $15.00 and date Sep 25, 2025 will parse with high confidence and go straight through to linking without bothering the user.

All parsed results, whether auto or manual, include a confidence score and a record of which parser (rule or ML model version) was applied, for auditing and future model improvement.

Linking

Once we have structured data from an email (e.g. a provider, amount, date, etc.), the next step is to **link** this information to the user's financial transactions in the internal ledger system. The goal is to match each parsed email (which represents a financial event) to the actual transaction recorded in the user's accounts. We use a **fuzzy matching algorithm** to account for slight discrepancies (dates, names) and ensure robust matching. If a match is found, we link the records; if not, we handle it by creating candidates or awaiting future transactions. We also link any attachments to the corresponding transaction for reference.

Fuzzy Matching Algorithm – Linking Emails to Ledger Transactions

Financial transactions in the ledger have attributes like date, amount, description (merchant name or reference), category, etc. The parsed email gives us at least an **amount and date**, and often a **provider name** or reference ID. We design a fuzzy matching algorithm that uses multiple identifiers and weighted scoring to find the best match in the ledger [22] :

1. **Candidate Selection:** For a given parsed email entry, gather candidate transactions from the ledger:
    - **By date:** Look for transactions whose date is the same as the email's date, or within a small window (perhaps ±1 day to account for posting delays or time zone differences).
    - **By amount:** Within those, find transactions with the exact amount. Amount is typically the most critical key – a transaction for $55.23 should match an email for $55.23, unless rare scenarios like split transactions. If no exact amount match is found on the exact date, widen the search slightly: same amount within a week range, or if still none, consider approximate amount if there's possibility of minor discrepancies (though usually receipts match exact amounts).
    - This yields a set of candidates that have matching amount and near date, or matching date and near amount in rare cases.
2. **Scoring:** For each candidate transaction, compute a similarity score:
    - **Amount match (weight 5):** If the amount exactly matches, add a strong weight (e.g. +5). If it's slightly off (which should rarely happen, maybe due to currency conversion or tips), we could handle that, but generally, we expect exact match for amount.
    - **Date match (weight 3):** If the transaction date matches the email date, score +3. If within 1 day, maybe +2 (to allow one-day differences), within 2-3 days +1 (though beyond that might not be considered).

- **Provider name match (weight 2):** Compare the provider/merchant from email with the transaction's description or payee. We use a fuzzy string match (case-insensitive, ignore punctuation, etc.). For instance, an email says "Apple Store" and the transaction says "APL*Apple Online". We would compute a similarity (for example, using Jaro-Winkler or Levenshtein distance). If above a threshold (say 0.8 similarity), consider it a match and score +2. If a very close match (or exact substring match), even higher. If no match or unknown, score 0 on this parameter.
- **Reference ID match (weight 2):** If the email provides a reference number (like an order ID or invoice number) and if our ledger transaction has a memo or reference field, we can check for that as well. A direct match of a reference ID would almost guarantee correctness, so that could be a high weight if available.
- (Optionally, **Category or type match**: If the email is clearly an expense vs income and the ledger transaction is labeled income vs expense, that should align. Usually amount sign could indicate that. This is minor since amount and date usually suffice.)

Each of these identifiers corresponds to an "agreement" or "disagreement" between the email and transaction [23]. We assign weights as above (tunable). The sum or weighted average produces an overall confidence that this transaction is the one corresponding to the email.

1. **Match Decision:** If one candidate has a score above a certain threshold (and higher than others by some margin), we deem it a match. For example, a perfect match might score 10+ (5+3+2, etc.), whereas the next candidate maybe scores 5. We would pick the top. If multiple candidates are close (e.g., two different transactions on the same day for the same amount), the system cannot be sure – in that case, we mark the linking as low confidence and send to manual review (user can select the correct transaction in the UI).

2. We aim to minimize **false positives** (linking to the wrong transaction) by requiring strong agreement on key fields [24]. If criteria are too broad, false positives can occur, so our matching must be strict enough. Conversely, if criteria are too narrow, we might miss true matches (false negatives) [24]. The weights/threshold are calibrated to balance this, and we prefer to err on asking the user in ambiguous cases rather than mislinking data.

3. **No Match Handling:** If no candidate transaction scores above the threshold or if the candidate set is empty (meaning no transaction in ledger with that amount & timeframe):

    - We create a **new "candidate transaction"** entry in a pending state. Essentially, we suspect this email corresponds to a transaction not recorded yet. This can happen if the user paid cash (so no bank transaction exists) or if the bank transaction hasn't posted yet (timing issue), or if it's an invoice for expected payment. We flag this to the user.
    - The new candidate transaction would include the details from the email (amount, date, provider) and perhaps be marked as an "unconfirmed" transaction in the ledger. The user can later confirm it or match it manually to a transaction that appears later.
    - If the ledger is updated from bank feeds, we could also implement a **delayed linking**: keep the parsed email around for a few days and keep trying to match if a new transaction shows up (especially useful if the email is a receipt and the card charge posts a day later). This can be implemented via a scheduled job that revisits unmatched parses for a period of time.

4. **Linking actions:** When a match is confirmed (automatically or via user selection), we update the relevant records:

    - The `parsedEntities` record (see Outputs) gets a pointer to the matched transaction's ID in the ledger.
    - The ledger transaction record gets an attachment/reference to the email (so from the accounting side one can see the source document).
    - We might also update category if the email provides one or if we deduce something (though category assignment is typically separate or user-defined).

By using this multi-identifier fuzzy matching, we ensure a robust reconciliation of email data to financial records. This is conceptually similar to data reconciliation systems which use multiple fields and scoring for entity resolution [22].

Handling Unmatched or New Transactions

If no ledger transaction was found to match an email (and we create a new candidate as described), how do we surface and resolve these? We will present "Unmatched email entries" in the user's interface (likely under the manual review section). The user can then: - **Confirm as new expense/income:** Maybe the email was a cash purchase or otherwise not tracked; the user can accept it and we will insert a new transaction in their ledger with appropriate details. (The system might prompt for which account to attribute it to, e.g. "Cash" account or a specific credit card if they forgot to log it.) - **Match manually:** The user might recognize that this email corresponds to an existing transaction (perhaps our algorithm missed because of naming differences). The UI could allow the user to pick from a list of recent transactions to link. Once linked, we update as if auto-matched. - **Dismiss:** If the email was not actually a financial transaction (or is a duplicate), the user can dismiss it, and we mark it as such (no further action).

We also consider the case of **incoming invoices (income)**. If the user receives an invoice for payment (e.g. they are a freelancer getting a bill to collect money), there might not be a transaction until the payment arrives. The system can still parse it and perhaps create a placeholder "accounts receivable" entry. If later a bank deposit of that amount appears, we can match them. If after some time no payment appears, we could remind the user or mark it as a pending income.

Linking Attachments to Transactions

Any attachments (like PDF invoices or images of receipts) carry valuable detailed information and serve as proof of the transaction. Once we have linked an email to a transaction (either existing or newly created), we will **attach the documents to that transaction record**. In practice, the ledger's data model should allow storing a link or ID of supporting documents. We will store the **URL or reference ID of the attachment** from our storage in the transaction's metadata.

The benefit is that later, in the accounting UI, the user can click on the transaction and view the original invoice/receipt image. Many modern accounting systems do this – for example, Receipt-AI automatically syncs the receipt image and data to QuickBooks, making reconciliation a one-click process [25] [26]. We aim for similar convenience.

If a transaction is matched to an email, then the email's attachments (PDF of an invoice, etc.) are effectively evidence for that transaction. We ensure they are accessible. If multiple attachments (maybe an email had two PDFs), all can be linked.

From an implementation standpoint, if using an internal ledger database, we might have a table for `transactions` and a table for `transaction_documents`. We would insert a record in `transaction_documents` linking the transaction ID to the `email_ingestion` attachment ID (or stored file path). This way, the data is normalized and we don't duplicate the file store.

Additionally, we propagate any **useful metadata** from the attachments. For instance, if an invoice PDF contains an **invoice number** or breakdown of items, we might include that in the transaction's memo or in the parsed data. That might not directly affect linking, but improves the richness of information stored.

In cases where a single email pertains to **multiple transactions** (rare, but for example, a monthly statement email might list multiple payments or a rideshare summary with multiple rides), our system should ideally split those in parsing and then attempt linking each amount separately. Our data model allows multiple parsed entities per email if needed, each linking to a different transaction.

Finally, linking is done in a way that's **auditable and reversible** – if a link is incorrect, the user (via the manual UI) can unlink the email from a transaction and either match it to a different one or create a new one. This flexibility is important for data integrity.

Outputs

After parsing and linking, the system produces structured outputs that represent the financial facts gleaned from emails. These outputs are stored and made accessible for use in the user's financial application (e.g. populating their expense ledger, budgets, etc.). Here we define the format of this structured data, how it's stored, and how we maintain traceability to the original emails.

Structured Output Format

Each parsed email (or each financial event within an email) is represented as a JSON-like structured object with key fields capturing the information. The canonical format will be:

```
{
  "provider": "...",       // e.g. "Amazon", "Starbucks"
  "amount": ...,           // numeric value, e.g. 45.67 (always positive; a
separate flag or context marks it as expense or income)
  "date":
"...",           // transaction date in ISO 8601 format (YYYY-MM-DD or full
timestamp if needed)
  "referenceId": "...",    // an identifier from the email, e.g. order number,
invoice number, confirmation code (if available)
  "category": "...",       // (optional) category of spending (e.g. "Food &
Dining") if we infer or user assigns it
  "transactionId": "...",  // (optional) internal ledger transaction ID if
```

```
linked
  "emailId": "...",        // internal ID or Gmail message ID referencing the
raw email
  "confidence": ...        // confidence score of the parse, e.g. 0.92
}
```

All fields except `provider`, `amount`, `date`, and `emailId` might be optional or null if not applicable. The **provider** is the merchant or source of the transaction. The **amount** is a decimal number (we can also include a currency code if multi-currency is supported; by default assume user's local currency or detect from content). The **date** is ideally the actual transaction date; if the email only had an order date or invoice date, we use that. **ReferenceId** gives an audit hook (like an invoice number or email's own ID) to trace back. If the item is linked to the ledger, `transactionId` will hold the ID of that ledger entry (for quick cross-reference).

We will store these structured outputs in a database table called `parsedEntities`. Each row corresponds to one JSON object as above (with columns for each field). Key columns: - `id` (primary key) - `userId` (link to user who owns it) - `emailId` (foreign key to `email_ingestion.id` of the source email) - `provider` - `amount` - `currency` (if needed) - `date` - `referenceId` - `category` - `transactionId` (nullable, foreign key to ledger transaction if matched) - `confidence` - `status` (e.g. "parsed", "reviewed", "confirmed") - `parserVersion` (maybe which parser or ML model version produced it, for audit)

By storing these in a table, we can easily query all parsed financial events, join with transactions, etc. This also decouples the parsing from the Gmail specifics – at this point the data is just structured financial info, which can feed into any UI or report.

We ensure to include a **reference link back to the raw email** for auditability and user trust. In the `parsedEntities` table, the `emailId` and possibly `emailMessageId` (Gmail's ID) provide that link. In the UI, we could use that to fetch and display the original email if needed (for example, a user auditing why a certain amount was recorded can click and see the source email content). Auditability is crucial – **every parsed data point should be traceable to its source document** [27]. We store the necessary references to achieve that.

Example Parsed Email and JSON Output

Let's walk through a concrete example to illustrate the output. Suppose the user received the following email:

Subject: Your Coffee Receipt
From: Starbucks receipts@starbucks.com
Date: Sep 26, 2025
Body:

```
Thank you for your purchase!

Store: Starbucks #1234 – Main Street
```

```
Date: 2025-09-26 10:15:22
Card: Visa **** 4242

Item                                    Price
-----------------------------------------------
Caffe Latte (Grande)                    $4.50
Pumpkin Muffin                          $3.75
-----------------------------------------------
Total: $8.25

Hope to see you again!
```

Our system would parse this email. A Starbucks-specific parser might recognize the format or our ML might detect the entities. The structured JSON output could be:

```
{
  "provider": "Starbucks",
  "amount": 8.25,
  "date": "2025-09-26",
  "referenceId": null,
  "category": "Food & Beverage",
  "transactionId": "TXN-100245",
  "emailId": 5567,
  "confidence": 0.98
}
```

Here, **provider** is "Starbucks". **Amount** is 8.25 (from “Total: $8.25”). **Date** is 2025-09-26. There’s no specific referenceId in this simple receipt, so it’s null (or could be the store # or last 4 of card, but those are not unique enough to be useful references). **Category** was assigned as "Food & Beverage" – perhaps our system mapped "Starbucks" to that category automatically or the user had set a rule; this field can also be filled in manual review if not auto. **TransactionId** "TXN-100245" indicates it found a matching transaction in the user’s ledger (maybe from their credit card) for $8.25 on that date, and linked to it. **EmailId** 5567 is the internal ID pointing to the raw email in `email_ingestion`. Confidence 0.98 means we’re very sure about this parse (likely a rule-based parse).

This JSON output is stored in `parsedEntities` and also can be delivered via API to the frontend if needed (for example, the Next.js app might fetch the user’s parsed receipts to display in a dashboard).

Storing Outputs and Reference Links

The final structured data is stored in the database as described, but it’s also important to consider how it’s used and referenced: - In the accounting UI, when listing transactions, we can show an icon or indicator if a transaction has a linked email/receipt. Clicking it can retrieve the `parsedEntities` entry and then allow viewing the email or details. - We maintain the **reference to raw email and attachments**: for audit. The `emailId` can fetch the original email and attachments from `email_ingestion`. We also might store directly in `parsedEntities` a field like `rawEmailLink` which could be a URL to Gmail (using the Gmail

message ID) if the user wants to view it in Gmail. (Gmail web interface can open a message by ID, if the user is logged in – we'd only provide this if it's useful and safe.) - Each `parsedEntities` record also includes which **parser processed it** (could be indicated by a parser name or version). This helps with debugging and with governance (knowing which version was used for each data point – important if a bug is later found in a parser, we know which entries might need rechecking).

Finally, after outputs are stored, they can be further utilized: e.g., the system could automatically categorize expenses, tally up totals by provider, etc., as part of the product's analytics. The **structured data unlocks a lot of possibilities** (budgeting, search by vendor, etc.).

All outputs are kept until the user deletes them or deletes their account. For privacy compliance, as discussed later, if a user requests deletion, these outputs and their source data must be wiped as well.

Ops & Governance

Building and maintaining Gmail Hub's parsing service requires robust operations and governance practices. We need to manage a growing library of parsers, ensure quality with each update, and uphold user privacy and compliance. Key areas include a **parser registry with versioning**, a **golden test set for regression testing**, and strict **privacy measures**. We also consider how the system can be rolled back or updated safely, and how we measure quality over time.

Parser Registry, Versioning, and Rollback

As we add more parser rules (DSL templates) and update the ML models, it's essential to track versions and allow quick rollback if something goes wrong. We will implement a **Parser Registry** – essentially a catalog of all parsing rules and models in the system: - Each **DSL parser template** is identified by a name and has a version number. For example, "AmazonOrderReceipt v1.0". If Amazon changes their email format, we might create a new version v1.1 of that parser to handle it. - The registry keeps the history of changes, who made them, and a description of what changed. - At runtime, the parsing engine can either use the latest version of each parser or even allow multiple versions concurrently (but usually we'd deploy the latest across the board, except perhaps during a migration/testing phase).

We adopt **version control** (like git) for parser definitions. Possibly the DSL files live in a repository; changes go through code review and testing. When deploying an update, we label a new version. The registry in the database might have entries like (parser_name, version, active_flag, etc.), enabling a quick switch back to an old version if needed.

For the **ML models**, versioning is equally important. Each trained model (NER model, etc.) gets a version identifier (e.g. "NER Model v2.1"). The code using the model should be aware of the model version. If a model update performs poorly unexpectedly, we should be able to revert to the previous model. Deployment of models might involve loading a different model file or endpoint. Containerization can help here: we might package parsers or models in containers tagged by version, so rolling back means running the older container image [28] . Azure's architecture references using versioned container images to ensure consistent deployment and easy rollback [28] – we can mirror that approach.

Testing and Rollback: Before promoting a new parser version to production (active use for all incoming emails), we run it against our test suite (discussed next). We might also run it in shadow mode on live data for a short period: i.e. parse emails with both old and new parser and compare results, without affecting the user, to ensure the new one works better. If any issue is detected after release (say users report mis-parses), we can either hotfix or roll back to the previous version quickly thanks to the versioning.

Our parser registry could also support **dynamic enable/disable** of specific parsers. For instance, if a certain parser is found to be causing errors, we could disable it (the engine will then ignore it and perhaps rely on ML for those emails) until it's fixed.

All these governance measures ensure that as our library scales to possibly hundreds of parser rules and numerous model iterations, we maintain reliability and can respond to problems swiftly.

Golden Test Set & Automated Validation

To guarantee that new changes don't introduce regressions, we will maintain a **golden test set** of emails and expected parsing outputs. This is essentially a collection of representative sample emails (covering all supported providers and various edge cases), each with a known correct structured result that we treat as ground truth.

Golden Set Composition: We will gather emails from various sources: - Synthetic or example emails from known providers (some providers offer sample receipts, or we can take actual receipts with sensitive info redacted). - Real user emails (with permission and anonymization) that cover tricky cases. - Edge cases like emails with missing fields, multi-currency, different languages (if we support international receipts, e.g. non-English, that adds complexity). - Both emails that hit DSL rules and ones that rely on ML.

For each, we store the email content and a manually validated JSON output (the expected parse). We might store this in a file or database, and possibly in our source repo so it's versioned along with parser code.

Automated Parser Validation: Whenever we update a parser or model, we run the entire golden set through the parsing pipeline (in a test mode) and compare the outputs to expected results. The system should report any mismatches: - If a previously correctly parsed field is now missing or wrong, that's a regression. - If the changes were intended (e.g. we improved something), then we update the expected output accordingly (after review).

We will integrate this into a CI (Continuous Integration) pipeline. No parser code change gets deployed unless the test suite passes 100% or any differences are understood and approved.

Additionally, we can incorporate **unit tests** for specific parser logic (for example, test that the regex in AmazonOrderReceipt parser indeed extracts a sample Amazon email's fields correctly). But the golden set acts as a high-level integration test which is very important for this kind of system.

Regression Example: Suppose we have a test email from Uber from 2024 and expected output with amount, date, etc. If an engineer changes the Uber parser to handle a new format but accidentally the regex no longer matches an older format, the golden test would show that the 2024 Uber email no longer parses correctly – alerting us to either handle both formats or version it.

We will also include a **continuous evaluation** mechanism. Similar to how Document AI monitoring works, we will track metrics on live parsing results: e.g. percentage of emails parsed without manual review, average confidence, etc., and any spikes in failures will trigger alerts. But offline, the golden set is our safety net before deployment.

Privacy & Compliance Measures

Handling users' financial emails demands strict privacy and compliance with regulations like GDPR. We implement several measures:

- **Data Minimization & Masking:** We only store data that is needed for the service. Raw emails can contain sensitive personal information (addresses, account numbers). We avoid logging raw content in plaintext in any application logs or error tracking. If we must log something for debugging, we mask PII (Personally Identifiable Information) such as email addresses, credit card numbers, etc. For example, if logging an error about "failed to parse card number 411111XXXX", we ensure the number is partially masked or not logged at all.
- **Encryption:** All stored email data and attachments are encrypted at rest (using database encryption or filesystem encryption). In transit, use HTTPS for any service communication. Since we are dealing with potentially sensitive financial info, encryption is essential to prevent leaks in case of a breach.
- **Access Control:** Internally, access to raw emails and parsed data is limited to services that need it. Developers or support personnel will not freely access user data without consent. If we provide a manual review UI, it is to the user themselves or authorized auditors – not an open system where staff randomly browse emails. If manual review by company staff is needed (perhaps if offering a bookkeeping service), that must be made clear to users and handled with least privilege.
- **User Consent and OAuth scopes:** We use OAuth to access Gmail, so the user explicitly grants our app permission to read their emails. We must ensure we request the minimal scopes (likely Gmail read-only or specific label access if possible). We also comply with Google's policies on data usage (no sharing, etc., beyond the app's function).
- **Data Retention and Deletion:** We honor user requests to delete their data (the "Right to be Forgotten"). If a user disconnects their Gmail or deletes their account with us, our system will delete all their ingested emails, attachments, and parsed outputs from our databases within a reasonable period. This includes removing any backups or at least rendering them inaccessible. We will design a **deletion workflow** that scrubs the user's rows from `email_ingestion`, `parsedEntities`, and any file storage. If some aggregated stats exist that included their data, those should be decoupled and not identifiable.
- **PII in outputs:** The structured outputs mostly contain financial info, but could include PII like provider names (which are usually companies, not personal info) or possibly if an email contains the user's name or address in an invoice, that might get parsed. We likely do not need the user's own personal info in outputs, so we avoid storing things like their address even if it was in an email. If needed, we might store a hash or partial if that's ever relevant. In general, **we treat financial data as sensitive** and apply similar protections as we would to PII.
- **Compliance & Security Audits:** We will follow best practices for securing cloud infrastructure. All access to Gmail Hub microservice is authenticated. We will also allow users to **opt out** of certain data uses – for example, if we ever wanted to use aggregated data to improve ML models, we would do so in a way that's compliant (e.g. anonymize it) or allow users to say no (especially under regulations for using their data for secondary purposes).

By implementing these privacy measures, we aim to protect user data. For example, if a user requests an export of their data or deletion, we have administrative tools to do so quickly. Masking PII also extends to how we might display data: e.g. if showing a portion of an email to a third-party service, we wouldn't include email addresses or full card numbers. Essentially, **user trust is paramount**, so we bake privacy into the design rather than tacking it on.

Additionally, we will undergo **periodic reviews** of our data handling. This includes verifying that we are not inadvertently storing any OAuth tokens or sensitive content unencrypted. If we integrate with any external services (like OCR APIs), we also ensure those comply with privacy standards (or use on-device OCR if privacy is a concern, albeit at possibly lower accuracy).

User Onboarding & AI Assistance

From the moment a user signs up, we want to streamline their experience through AI-driven assistance. The Gmail Hub leverages an **AI onboarding agent** to interact with users in natural language, guiding them to set up and providing context that improves the system's accuracy. This agent essentially "interviews" the user about their financial context and preferences, and continues to serve as a helper for ingesting additional data (like receipts via other channels) after the initial onboarding.

AI-guided Onboarding Interview

After a user connects their Gmail account, they may be presented with a conversational UI (chatbot-like, possibly integrated into our Next.js frontend) for onboarding. The AI agent (powered by an LLM or a scripted decision tree with AI components) will cover topics such as: - **Financial Accounts & Context:** The agent might ask which bank accounts or credit cards the user uses for most transactions. This could help because if we know the user primarily uses, say, a Visa ending 4242, we might later auto-link receipts to that card's transactions and ignore others. It could also ask if they use multiple currencies or have businesses separate from personal finances. - **Frequent Providers or Bills:** The agent could inquire about any regular bills or subscriptions the user has (e.g. "Do you receive regular invoices or bills we should look out for?"). If the user says "Yes, I get a monthly invoice from ACME Corp for consulting work," we can ensure to parse those and maybe set a rule to categorize it appropriately. - **Categories & Budget Prefs:** The user might be asked if they follow a certain expense categorization (maybe they use standard categories like Food, Travel, etc. or custom ones). The agent could even show a list of categories and let the user tweak or choose which they care about. This info will help when we categorize parsed transactions. For example, the user can tell the agent "All Starbucks purchases should be categorized as Coffee" – the system then knows to label that provider accordingly going forward. - **Historical Data Range:** The agent can confirm how far back to ingest emails. Perhaps the user doesn't want 10 years of history, just the last 2 years. The agent can set that preference which the ingestion process will use.

This interview makes onboarding interactive and personalized, rather than a dull form. With AI, the agent can understand free-form responses. For example, the user might say, "I have two credit cards and a checking account, I mostly use the Amex for travel expenses." The agent can parse that and maybe set an internal note that "Amex card is used for travel – perhaps categorize flights/hotels differently" or at least be aware of an Amex card (if the email receipts show an Amex, it knows that's the user's card).

Establishing Trust: The AI will also explain to the user what the system will do – e.g. "I'm going to scan your inbox for receipts and bills. I'll categorize them and match to your transactions. Don't worry, I only look at financial emails, not personal ones." This sets expectations and assures privacy.

Continuing to Accept Receipts Post-Onboarding

After initial setup, users will continue to accumulate receipts and documents. The Gmail Hub system primarily watches their Gmail for new emails. But what about receipts that aren't in email form? For example, a paper receipt from a taxi, or a screenshot of an online payment confirmation. The system should allow those to be ingested as well: - **Email Forwarding:** We can instruct users that if they have an email in another account or any email content, they can forward it to their connected Gmail (or perhaps a special address we provide). Since our system is reading their Gmail, any forwarded receipts will be picked up. Alternatively, we could provide an **inbound email address** (like user+receipts@ourapp.com) where they can send receipts directly – but that may complicate ingestion since that's outside Gmail. Simpler is to rely on their Gmail; many users forward receipts to themselves anyway. - **Direct Upload / Photo Capture:** The frontend could have a feature to upload a receipt image or take a photo (especially since our frontend stack is React/Next.js, we can integrate a file uploader or mobile camera access). When a user uploads an image of a receipt, we feed it into the same ingestion pipeline: the image is sent to the backend, an OCR is run (similar to email attachments OCR), and then parse it. The AI agent can be integrated here to confirm details if needed. - **SMS / Chatbot**: While not explicitly required, one could extend to interfaces like texting a receipt photo, similar to Receipt-AI's approach [29], but that's an additional channel. For now, focusing on Gmail and direct upload covers most cases.

The AI agent plays a role post-onboarding as well: - If a new provider is encountered that the system isn't sure about, the agent could proactively ask the user: "We saw a receipt from **Bob's Auto** for $200. Is this a new expense for your car? What category would you like to assign to it?" This can be done via a chat message or notification. The user's answer ("Yes, that's Car Maintenance") can then train the system to categorize future similar ones and possibly update the parser knowledge (like linking Bob's Auto to category Auto Maintenance). - The agent can also help with **disambiguation**. Suppose two transactions matched one receipt equally well. Instead of just putting it in a manual queue silently, the agent might message: "I have a receipt for $100 on 2025-09-20 but I see two $100 transactions around that date (one on 09-19 at Supermarket, one on 09-21 at Gas Station). Which does this receipt belong to?" The user can respond, and the system links accordingly. This makes manual review more conversational.

- The agent can remind the user to forward or add receipts that might be missing. For example, if the user mentions in conversation or if the system detects a transaction in the bank with no corresponding receipt, the agent could ask "Do you have a receipt for X purchase? If so, you can forward it to your Gmail and I'll catch it."

Technical Implementation: The AI agent can be built using a conversational AI platform or even simple logic with an LLM. It will likely require maintaining context about the user's data (account names, categories, any missing info). We must ensure the agent's suggestions are accurate; it might use the parsed data and linking results to generate its questions. The Next.js front end would have a chat interface component that interacts with a backend endpoint (which in turn uses an AI model or rules to respond).

Given our tech stack, we could integrate something like OpenAI's API for natural language understanding. But we'd be cautious to not send sensitive data in prompts without user consent. Alternatively, a rules-

based approach for the initial interview (with multiple-choice or form filling aided by a bit of NLP) might suffice.

User Experience: The goal is to make the typically tedious onboarding (connecting accounts, configuring categories) feel like a friendly conversation. The agent can also educate: e.g., "Great, I've categorized Starbucks under Coffee. In the future, you can ask me 'What did I spend on coffee this month?' and I can tell you." – hinting at possible query features.

Overall, the AI assistant helps personalize and continuously improve the system's understanding of the user's finances, resulting in more accurate parsing and linking.

QA and Evaluation Metrics

To ensure Gmail Hub's ongoing quality and to quantify its performance, we establish a comprehensive QA and evaluation plan. This includes metrics to evaluate parsing accuracy, linking success, system throughput, and user satisfaction. We will monitor these metrics and use them to guide improvements.

Parsing Accuracy Metrics

We will measure the performance of the parsing stage in terms of **precision** and **recall** for entity extraction: - **Precision**: Out of all the fields the system extracted, what percentage are correct? For example, if it parsed 100 amounts and 95 were exactly correct, precision for Amount is 95%. High precision is crucial so that we don't output incorrect data. - **Recall**: Out of all the fields that *should* have been extracted (according to ground truth), what percentage did the system capture? If 100 emails had an amount, and we successfully extracted 90 of them, recall is 90%. We want good recall so we're not missing data, though we often prioritize precision a bit higher in finance (better to ask the user than to guess wrongly). - We will compute these metrics per entity type (amount, date, provider, etc.) using our **golden test set** where ground truth is known. We'll also periodically sample real production data (with user permission or using feedback as pseudo-ground truth) to see how we're doing in the wild.

Additionally, we use an **overall parse success rate**: percentage of emails that were fully parsed (all key fields) without needing manual intervention. We expect this to improve over time as we add more templates and train models on more data.

Linking Success Metrics

For the linking stage, we define: - **Auto-Match Rate**: The percentage of parsed emails that were automatically matched to a ledger transaction (without user input). This reflects how well our fuzzy matching is working. If this rate is low, users are having to manually match too often. - **Match Precision**: We need to be careful that when we auto-link, it's correct. We might measure this by auditing a sample of auto-matches to see if any were wrong. Ideally, user feedback helps here: if a user disconnects a link or corrects it, that indicates a false positive match. We track those instances. Our aim is to keep false linkages near zero. - **Unmatched Rate**: How many parsed emails end up with no match (requiring new transaction creation)? If high, it might mean we should improve matching logic or it might reflect a lot of outside-of-ledger transactions (like cash expenses). We can use this to perhaps prompt users to integrate more accounts if they have many unmatched (e.g. "We found 20 receipts that didn't match any bank transactions – do you have another account these might belong to?"). - **Time-to-Link**: If we are waiting for future

transactions for some receipts (like invoices awaiting payment), we could measure how long on average until they get matched or confirmed. This is more operational, but could inform if our waiting strategy is effective.

Manual Review and User Feedback

We will use **user feedback loops** as an evaluation tool: - Track the number of items routed to manual review and what percentage of those the user modifies vs. just confirms. If users frequently have to change the parsed data, that signals specific weaknesses to fix (maybe a particular provider always has wrong extraction). - **User correction patterns** are valuable data. Our system will log what corrections are made (in aggregate, without exposing PII) to feed back into improving the DSL rules or retraining the ML model [30]. For instance, if we see that in 50% of "ShopXYZ" emails, the user had to fix the date, perhaps our parser is misidentifying the date field and needs adjustment. - Survey or NPS-like feedback from users about the feature can be a metric too (though qualitative). E.g., asking users "How accurate have your email-derived transactions been?" and tracking that.

Performance and Operational Metrics

While accuracy is key, we also evaluate system performance: - **Latency**: How long from an email arriving to it being parsed and linked? If using push, this might be just a few seconds. We can measure the end-to-end latency distribution. The goal is to keep it low so that, for example, if a user gets an email and immediately checks their finance dashboard, the info is already there. We might set a target like 95% of emails processed within 1 minute of arrival. - **Throughput/Scalability**: We monitor how many emails per hour we can handle, especially during the initial onboarding of thousands of emails. We ensure our microservice can scale horizontally if needed. Metrics like CPU/memory usage, queue lengths (if using a queue system for parsing tasks), and error rates will be watched. - **Error rates**: Any failures (e.g., Gmail API call failures, parsing exceptions) are tracked via logs. We aim for a high success rate in processing. If errors occur (say OCR fails for a certain image type), those are flagged and categorized.

Regression Testing and Continuous Evaluation

As mentioned, the **golden test set** gives us a regression test metric: we expect 100% pass on that for any production release. We'll augment this with periodic **re-evaluation of models**. For instance, if we develop a new ML model version, we will compare its precision/recall on a validation set to the old model's performance and ensure it's an improvement or at least not a regression. Google's Document AI provides precision/recall metrics for evaluation [31], and we would similarly compute F1-scores for our model and aim to increase them with each iteration.

We also track **confidence score distribution**. If we see a lot of parses with borderline confidence (e.g. many around 0.6-0.7), maybe our threshold or model calibration needs adjustment. The goal is to push more items into high confidence (through improvements) so that the manual queue is minimized over time.

Security and Compliance Checks

As part of QA, we'll periodically do audits to ensure compliance: e.g., verify that the deletion process indeed removes all data (perhaps part of QA test to simulate a user deletion and then attempt to find any of their data remaining). While not a metric per se, it's part of the quality governance.

Summary of Key Metrics:

- Parse Precision/Recall per field (target: e.g. >95% precision on Amount and Date).
- Percentage of emails fully parsed with no manual intervention (target: increasing over time, say 70% then 80%...).
- Auto-link rate vs. manual link rate (target: majority auto-linked if possible).
- False match count (target: extremely low; each false match is addressed).
- Manual review count and correction frequency (should decrease over time).
- Turnaround time per email (keep low).
- System uptime and error-free processing percentage (reliability).

We will create **dashboard reports** for these metrics (the backend can log events to an analytics database or monitoring tool). For example, a Power BI or Grafana dashboard could show daily how many emails processed, how many auto vs manual, average confidence, etc. Azure's architecture mentions monitoring such KPIs and user corrections for optimization [30] – we will implement a similar analytics pipeline to continually observe and improve our service.

By rigorously tracking these metrics and continuously testing, Gmail Hub's financial email intelligence service will remain accurate, reliable, and responsive to users' needs, even as email formats evolve or user circumstances change.

Sources:

- Gmail API push vs polling and Pub/Sub webhook usage [1] [2] [7] [9].
- Real-world Gmail push reliability considerations [10].
- Storing raw emails and attachments (binary data and blob storage) [11] [12].
- OCR and parsing of invoice attachments [13].
- Identifying invoice emails by keywords and senders [14].
- AI extraction of key invoice fields using learned patterns [19].
- Confidence-based routing for human review [20] [21].
- Fuzzy matching using multiple weighted criteria [22] [24].
- Linking receipts to accounting records for easier reconciliation [25] [26].
- Audit trail and traceability of parsed data [27].
- Deployment versioning and rollback of parsers/models [28].
- Continuous monitoring of pipeline metrics and user corrections [30].

---

1 7 8 9 Push Notifications | Gmail | Google for Developers
https://developers.google.com/workspace/gmail/api/guides/push

2 5 6 15 16 18 Gmail API Essentials
https://rollout.com/integration-guides/gmail/api-essentials

3 4 10 Gmail API's push notifications bug and how we worked around it at Hiver | by Raghav C S | Hiver Engineering | Medium
https://medium.com/hiver-engineering/gmail-apis-push-notifications-bug-and-how-we-worked-around-it-at-hiver-a0a114df47b4

11 What data type to store raw IMAP fetched email messages in Postgresql? - Stack Overflow
https://stackoverflow.com/questions/45175721/what-data-type-to-store-raw-imap-fetched-email-messages-in-postgresql

12 20 27 28 30 Extract and Map Information from Unstructured Content - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/ai-ml/idea/multi-modal-content-processing

13 14 19 A Guide to Email Invoice Parsing
https://tailride.so/blog/email-invoice-parsing

17 Synchronizing Clients with Gmail | Google for Developers
https://developers.google.com/workspace/gmail/api/guides/sync

21 Custom extractor overview | Document AI | Google Cloud
https://cloud.google.com/document-ai/docs/custom-extractor-overview

22 23 24 Fuzzy Matching 101: Cleaning and Linking Messy Data - Data Ladder
https://dataladder.com/fuzzy-matching-101/

25 26 29 Texting Their Way to $1.5K/Mo - receipt-ai.com
https://www.wearefounders.uk/case-study-texting-their-way-to-1-5k-mo-receipt-ai-com/

31 Evaluate performance | Document AI - Google Cloud
https://cloud.google.com/document-ai/docs/evaluate