USER GUIDE & SYSTEM DOCUMENTATION

EmailInsightAI

A supervisor's console for company mailboxes. It pulls every employee's mail through Gmail or Microsoft 365, asks Gemini what each message means, and turns the answers into counts a manager can act on — who is waiting for a reply, who is unhappy, what is a sales or AMC thread, and what is just noise.

`Laravel 12 · PHP 8.2+` `MySQL master + per-client databases` `Google Workspace & Microsoft 365` `Gemini 2.5 Flash → Pro fallback` `Blade · Bootstrap 5 · jQuery`

01

What the product does

EmailInsightAI is not a mail client that replaces Gmail or Outlook. It is a **monitoring layer over the mailboxes of a team**. A company admin registers their employees' addresses once; from then on a background job reads new mail for every address, classifies it with Gemini, and stores the result. Admins and department heads then open the console and work from the classification rather than from the inbox.

`Collect`

Mail arrives on its own

A scheduled job fetches new messages per mailbox using a Google service account or an Azure app registration. No employee has to install anything.

`Classify`

Gemini reads each message

Every message gets a sentiment, an action-required verdict, and a business category (sales, AMC, external), stored alongside the raw mail.

`Act`

Counts you can click

Reply-pending, replied, positive, negative, neutral, sales, AMC, promotion — every number in the toolbar opens the exact list behind it.

`Report`

A daily digest by mail

Each afternoon every admin and department head receives one table: a row per employee, a column per signal, every cell a deep link into the console.

The one thing to understand first

Almost every screen in the console is scoped to a **mailbox owner**, not to the person logged in. When you open a filter as an admin you are looking at *your own* mail; when you click through from the employee list, the same screen shows *that employee's* mail, and the employee's id travels in the URL (`/mails/send-mail/42`). If a screen looks empty, check whose mailbox you are in.

02

Accounts, roles & access

There is a single `users` table per company, and three fields decide what a row can do: `user_level` (what they may open), `auth_id` (which admin created them), and `account_type` (which mail provider to talk to).

ROLE	USER_LEVEL	CAN SIGN IN?	SEES THE MAIL OF
Company admin	2	Yes	Every user they created (`auth_id = their id`) plus themselves
Department head	1	Yes	Users assigned to them (`Department_head_id = their id`) plus themselves
Employee	0	No — login is refused with *"Only Admin Or Department Head Have Access"*	Nothing. Their mailbox is monitored on their behalf.

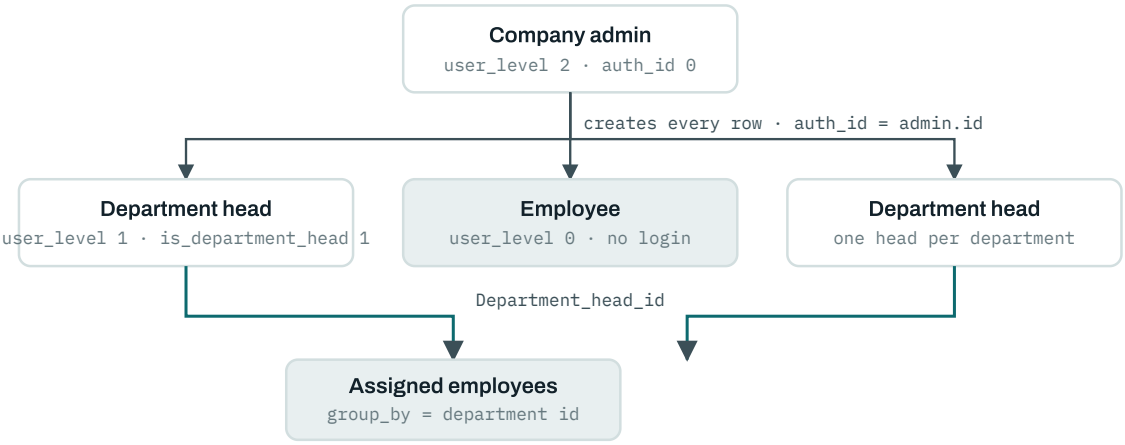

Fig. 1 — Every user row is created by an admin, so `auth_id` defines the company boundary. A department head only widens their own view: the console filters employees by `Department_head_id`, and one department may hold exactly one head.

Provider type

`account_type` decides which API is used for that mailbox, and it is chosen at registration:

- **0 — Google Workspace.** A service account with domain-wide delegation impersonates the mailbox. The key lives at `storage/app/email/service-account.json`; reading uses the `gmail.readonly` scope and replying uses `gmail.modify`.
- **1 — Microsoft 365 / Outlook.** An Azure AD app registration takes a client-credentials token for `graph.microsoft.com/.default`, cached in `storage/app/email/outlook_account.json`, and calls Microsoft Graph as the user.

> WHY REGISTRATION CAN FAIL
>
> Registering an address does more than insert a row: the app immediately tries to reach that mailbox. If Google or Azure refuses the impersonation you get *"You Are not Company Team member"* and no user is created. That is the intended gate — only mailboxes inside the delegated domain or tenant can be added.

03

Registering and signing in

Registration

01 Open `/register` and pick the account type — Google Workspace or Outlook.

02 Enter the mailbox address and a password (twice).

03 On submit, the app validates the address is unique, then tests the provider connection for that exact mailbox.

04 If the connection succeeds the account is created as a level-2 admin and you are returned to the form with a success message; if not, nothing is saved.

Signing in — password, then a one-time code

Login is two-step. The password is checked first but *no session is opened yet*; a six-digit code is generated, stored in `user_otps` with a five-minute expiry, and emailed. Only after the code verifies is the session created.

01 Enter email and password at `/login`.

02 Level-0 employees are stopped here by design.

03 A code arrives by mail, sent through Gmail SMTP as *"EmailInsightAI — One Time Password for Secure Login"*.

04 Enter it on the OTP screen. **Resend** issues a fresh code and invalidates the old one.

05 The code is deleted on use and you land on the dashboard.

> NOTE FOR OPERATORS
>
> Every OTP mail is also CC'd to a fixed internal address in `app/Http/Controllers/AuthController.php`. If that is a leftover from testing, remove the CC before the system carries real customers — see Findings.

Forgotten passwords, and first password for a new head

- **Forgot password.** `/forgot-password` writes a 64-character token to `password_reset_tokens` and mails a link to `/reset-password?token=…&email=…`.
- **New department head.** When an admin promotes someone to head, the app issues a reset token automatically and mails them a *"You're Assigned as Department Head — Set Your Password"* link pointing at `/set-password`. That link is how a head gets their first working password.

The dashboard

`/dashboard` is deliberately thin. It confirms which mailbox you are signed in as, shows a Google or Microsoft glyph for the provider, and re-tests the provider connection on every load. If that test fails you see *"Please try Later"* instead of the action buttons — that message means **the provider connection is down or the credential file is missing**, not that your password is wrong.

04

The mail workspace

Email Inbox in the top navigation opens `/mail-info`. Two things happen on load: the page triggers a fetch for new mail for that mailbox, and it refreshes the toolbar counts from `/mail/stats`. The toolbar is the product — each chip is both a number and a link to the list behind it.

CHIP	OPENS	WHAT LANDS HERE
Reply ▸ Pending Reply	`/mails/send-mail`	Gemini said a response is needed and no reply has been detected in the thread yet. `act on this`
Reply ▸ Replied Mail	`/mails/replied_mail`	Needed a response and a later message in the same thread was found. `closed loop`
Group By ▸ Sales	`/mails/sales`	Business pitches, quotations, leads, proposals — not AMC.
Group By ▸ AMC	`/mails/amc`	Annual maintenance contracts: renewals, service visits, SLA, warranty.
Group By ▸ *your groups*	`/mails/search-{id}`	Every sender you filed into that inbox group (see §08).
Sentiment ▸ Positive	`/mails/is_not_spam`	Appreciation, thanks, politeness. `positive`
Sentiment ▸ Negative	`/mails/is_spam`	Complaints, blame, frustration. `negative`
Sentiment ▸ Neutral	`/mails/neutral`	Status updates and factual statements. `neutral`

CHIP	OPENS	WHAT LANDS HERE
Promotion	`/mails/promotion`	Newsletters, offers, campaigns — anything Gmail labelled Promotions, Social or Updates.
Unread	`/mails/unread`	Not yet opened in the console.
Read	`/mails/read`	Already opened in the console.
All Mail	`/mails/all`	Everything stored for the mailbox.
Summary	`/fetchsummary`	Opens the AI digest modal (admins only) — see §06.

RULES EVERY LIST OBEYS

- Your own address is never shown as a sender — you do not see your own outbound mail as inbox items.
- Results are de-duplicated by provider message id, so one message appears once even if stored more than once.
- Anything Gmail put in **Spam** is excluded from every signal count.
- Promotions are excluded from sentiment, reply and category counts — they only appear under **Promotion**.
- Sentiment, category and promotion lists show **unread** items only; the reply lists ignore read state.

Each row in a list shows sender, subject, date, sentiment, label and action-required. On the reply-pending list the date column becomes **Waiting for Reply Since**, and the longer a message has waited the stronger the pill it gets — that is the intended way to triage a backlog.

05

Reading and replying

Opening a message

View on any row opens `/mails-show/{message}/{is_sent?}/{user?}`. The stored HTML body is rendered with invisible-character cleanup applied (zero-width joiners, soft hyphens and similar are stripped so pasted content does not break layout). Attachments are flagged by the `attachment_status` column.

Replying from inside the console

Reply opens a compose form pre-filled from the original message. Sending goes out through the provider as the mailbox owner, in the original thread:

- **Google.** A fresh service-account token with the `gmail.modify` scope is minted for that mailbox, then a MIME reply is posted to the Gmail API carrying the original `Message-ID` and `threadId` so it threads correctly.
- **Outlook.** Microsoft Graph creates a reply draft on the original message and sends it, which keeps Graph's own threading intact.

How "Replied" is decided

Nothing marks a thread replied when you press send. The determination happens on the *next* ingestion run, and it is worth understanding because it explains why a reply can take a few minutes to show up in the counts.

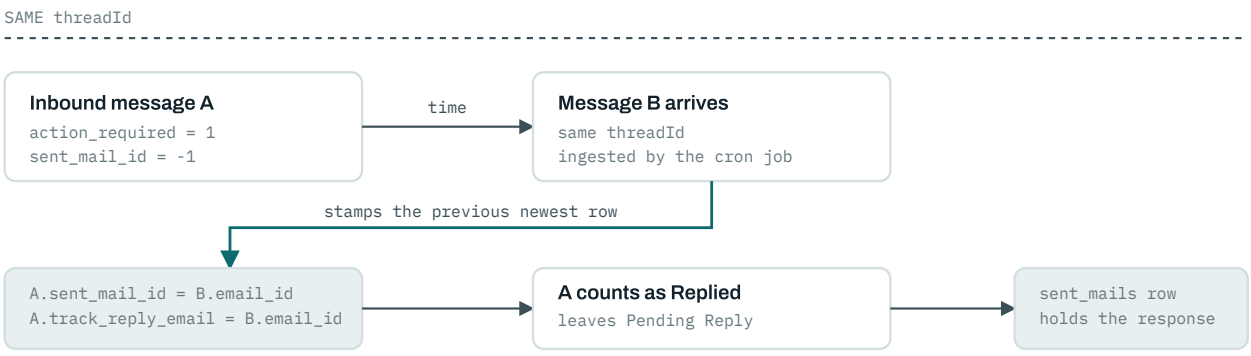

Fig. 2 — Reply detection is inferred from thread continuation, not from the send action. Until the next ingestion run sees message B, message A stays in `Pending Reply`.

> CONSEQUENCE WORTH KNOWING
>
> Any later message on the thread closes the loop — including one from the original sender. The counts measure *thread movement*, which is close to but not identical to "somebody on our side answered".

06

On-demand AI summary

The **Summary** chip is visible to admins. It takes the twenty most recent stored messages for the mailbox, excluding promotions and anything Gmail marked spam, joins each to its recorded response if one exists, and sends the batch to Gemini with instructions to return grouped HTML — newest thread first, with sender, recipient, date, subject, a one-line insight and a fuller summary covering actions, decisions, deadlines and attachments.

Endpoint	`GET /fetchsummary?user_id={id}`
Model	`gemini-2.5-flash`, falling back to `gemini-2.5-pro`
Scope	Latest 20 messages, promotions and Gmail-spam excluded
Output	HTML rendered directly into the modal

What you may see instead of a summary, and what it means:

MESSAGE	MEANING
"No messages found for this user"	Nothing eligible is stored yet, or everything stored is promotional. Fetch first.
"App Password is Expire..."	The mailbox row is flagged `is_login = 1`. The provider connection needs fixing before a summary can be built.
"Rate limit or other error"	Gemini returned 429 or 503. Both models were tried; wait and retry.

A separate experimental page at `/mic` offers speech input, and `public/dialogflow.php` is a Dialogflow CX webhook that forwards a spoken question to Gemini and returns the answer. Neither is wired into the mail console — treat them as prototypes.

07

Adding mailboxes & teams

Bulk upload — the normal way to onboard

Add Email opens the bulk import screen. Download the sample workbook first; the importer checks the header row before reading anything else.

01 Click **Download Sample** and fill it in. Required columns, in order: `First Name`, `Last Name`, `Email Id`.

02 Upload the `.xlsx`. A mismatched header stops the import and tells you exactly which columns were expected.

03 Every row is tested against the mail provider before it is accepted.

04 Read the result table. Rows are sorted into four buckets and each bucket can be exported back to Excel:

BUCKET	WHY A ROW LANDS THERE	WHAT TO DO
`Valid`	Provider accepted the mailbox; user created and queued for its first fetch.	Nothing.
`Blank`	Missing email address in the row.	Fill it in and re-upload just those rows.
`Error`	Provider refused — address outside the delegated domain or tenant, or a typo.	Check the address, then the delegation scope.
`Duplicate`	That address already exists in `users`.	Nothing — it is already monitored.

> WHAT A VALID ROW ACTUALLY CREATES
>
> A level-0 user owned by you (`auth_id` = your id), with a default password and the provider type copied from your own account — **plus a high-priority job in the master queue**, so the new mailbox is picked up on the next run instead of waiting for tomorrow's seed.

The employee list

All Employee shows one row per monitored mailbox with its live counts: total, positive, negative, neutral, reply-pending, replied and promotion. Clicking a number opens `/employee_process/{id}/{type}` — the same list, scoped to that employee and that signal. A stored counter (`freshcounts`, keyed by admin) is polled in the background so the page can flag when new mail has landed for someone since you last looked.

Departments and heads

01 **Settings ▸ Add Department** — create departments with a name and a label. Names and labels must both be unique. Departments flagged non-deletable cannot be removed.

02 **Settings ▸ Assign Department** — pick a department, choose one head, tick the employees who report to them. The head is promoted to level 1 and emailed a set-password link; each employee gets `Department_head_id` and `group_by` set.

03 **Settings ▸ Department Group** — review heads per department and add more people under an existing head.

04 **Remove** on a member clears their head assignment. If that leaves the head with nobody, the head is demoted back to level 0 automatically.

> ONE HEAD PER DEPARTMENT
>
> The system refuses a second head for the same department: *"One Department Contain only one Head Please Select Another Department"*. To move headship, remove the current head's members first.

Tuning what you see

Sender importance — Email Settings

Every new sender is added to a per-mailbox sender list automatically the first time they mail you. **Email Settings** splits that list into *Important* and *Not Important*, with per-sender counts of total, positive and negative messages. Tick the senders that matter and save: the ticked ones become important and **every unticked sender is marked not-important in the same action**. Selection is absolute, not additive.

Show all mail next to a sender lists every message stored from that address — the fastest way to audit one correspondent.

Inbox groups

Groups are saved sender sets that appear as extra entries under **Group By**.

01 Create a group under **Inbox Groups**.

02 From the sender list, assign senders to it.

03 The group shows up in the Group By dropdown as `/mails/search-{group_id}`, listing mail from all its senders.

04 Deleting a group releases its senders rather than deleting them.

Domain — internal versus outside

Domain stores one domain per user (default `@gmail.com`) and splits your sender list into matching and non-matching, with message counts on each side. Set it to your company domain to see at a glance how much traffic is internal and how much comes from outside.

Prompts and keywords

The prompt store lets an admin save a custom AI prompt (up to 200 characters) and a keyword label (up to 25). Saving a new one retires the previous active entry, so exactly one is live at a time. The active keyword is used as a heading for the positive-mail count in the console and in the daily report.

> CURRENT LIMITATION
>
> The prompt store is wired up end-to-end (`POST /prompt-form`, the `prompts` table, the keyword shown as a heading), but the classifier prompt itself is hard-coded in the ingestion command — a saved custom prompt does not yet change how mail is classified.

09

The daily summary email

Every admin and every department head receives a **Daily Summary** mail listing the people they are responsible for. One row per mailbox, one column per signal, and every cell links straight into the matching filter in the console.

COLUMN	COUNTS
Positive	Unread positive-sentiment mail, promotions and spam excluded.
Negative	Unread negative-sentiment mail.
Neutral	Unread neutral-sentiment mail.
Reply pending — 1 month	Action required, no reply detected, received within the last month.

COLUMN	COUNTS
`Reply pending — 48 hrs`	Action required, no reply detected, received in the last 48 hours.
`Replied`	Action required and a reply was detected.
Sales	Unread mail categorised as sales.
AMC	Unread mail categorised as AMC.

Command	`php artisan app:send-daily-email-summary`
Schedule	Every five minutes between 13:00 and 18:00 server time
Work source	The `cron_mail_send` queue in the master database, five clients per run
Transport	PHPMailer over Gmail SMTP, STARTTLS on port 587

Because the queue is claimed per client and marked done, each client is mailed once per day even though the command runs many times inside the window. Sending is retried on the next tick if a client's job fails.

10

How classification works

Ingestion and classification are one background command, `app:process-email-queue`, scheduled every ten minutes. It works through a queue of jobs — one job per mailbox per day — five at a time.

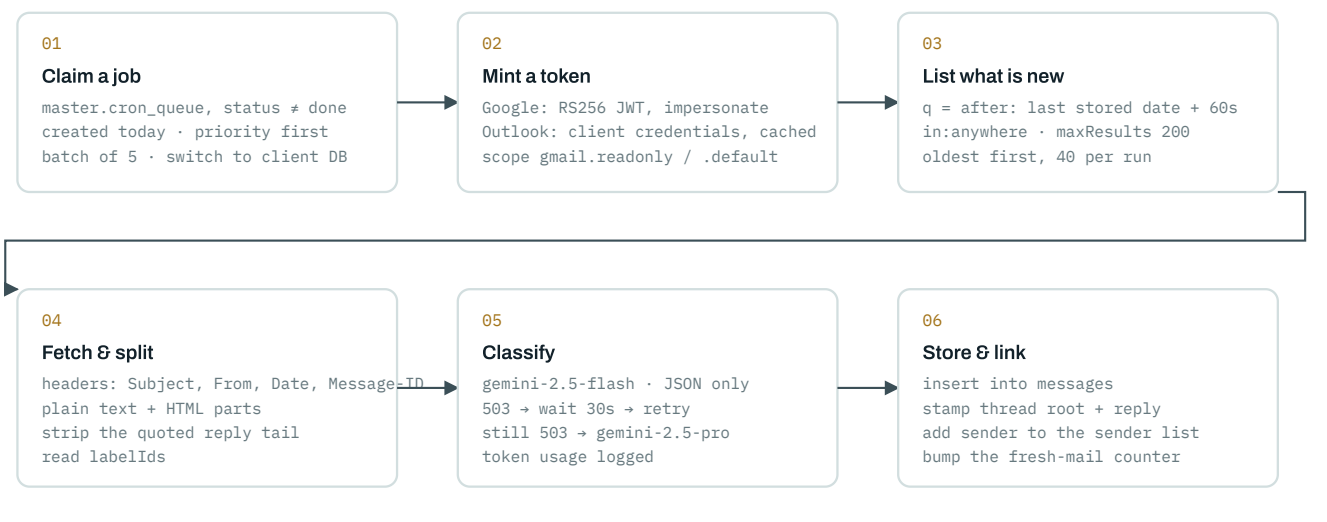

Fig. 3 — One pass of `app:process-email-queue`. The window is anchored on the newest message already stored, so a run only ever asks the provider for genuinely new mail.

What Gemini is asked

The classifier prompt instructs the model to read **only the new reply portion** of a message — quoted history and "On … wrote:" blocks are explicitly excluded — and to answer with JSON only, always returning every field even when uncertain:

- `action_required` — `send-mail` if the message asks a question, requests help or needs a response; otherwise `no-action`.
- `sentiment` — `positive`, `negative` or `neutral`.
- `category` — `amc` for maintenance contracts, renewals, SLA and service visits; `sales` for pitches, quotations and leads that are not AMC;

`external` for press, media and everything else.

- `mass_mail` — Outlook mailboxes also get `promotion` / `transactional` / `other`, because Graph has no equivalent of Gmail's Promotions label.

How the answer is stored

The raw JSON is kept in `gemini_return_json` and the words are kept in `is_spam_value`, `is_action_required_value` and `is_category_value`, so a classification can always be audited. Alongside them the answer is reduced to numeric codes that the filters query.

COLUMN	VALUE	MEANING
is_spam sentiment code	1	positive
	0	negative
	2	send-mail
	3	neutral
	4	promotion
	5	sales
	6	amc
	101	unrecognised answer — the model returned something outside the list
action_required	1	a reply is needed
	0	no action
	101	unrecognised answer
category	1	sales
	2	amc
	3	external
	101	unrecognised answer
is_promotion	1	Gmail labelled it CATEGORY_PROMOTIONS, CATEGORY_SOCIAL or CATEGORY_UPDATES
	0	not promotional
sent_mail_id	-1	no reply detected on the thread yet
	message id	the later message that closed the loop

> READING A 101
>
> A `101` in any of these columns is a classification miss, not a category. If they cluster, inspect `gemini_return_json` for those rows — it usually means the model wrapped its JSON in prose or the message body arrived empty.

Promotion detection is label-driven for Gmail and model-driven for Outlook. Messages carrying Gmail's `TRASH` or `DRAFT` labels are neutralised so they cannot appear as inbound mail, and `SENT` messages are flagged as outbound rather than counted as received.

11

System architecture

One Laravel application serves both halves of the system: an HTTP half that renders the console, and a console half that runs the scheduled ingestion and reporting. Both talk to the same providers, and both switch database connections at runtime depending on which company they are serving.

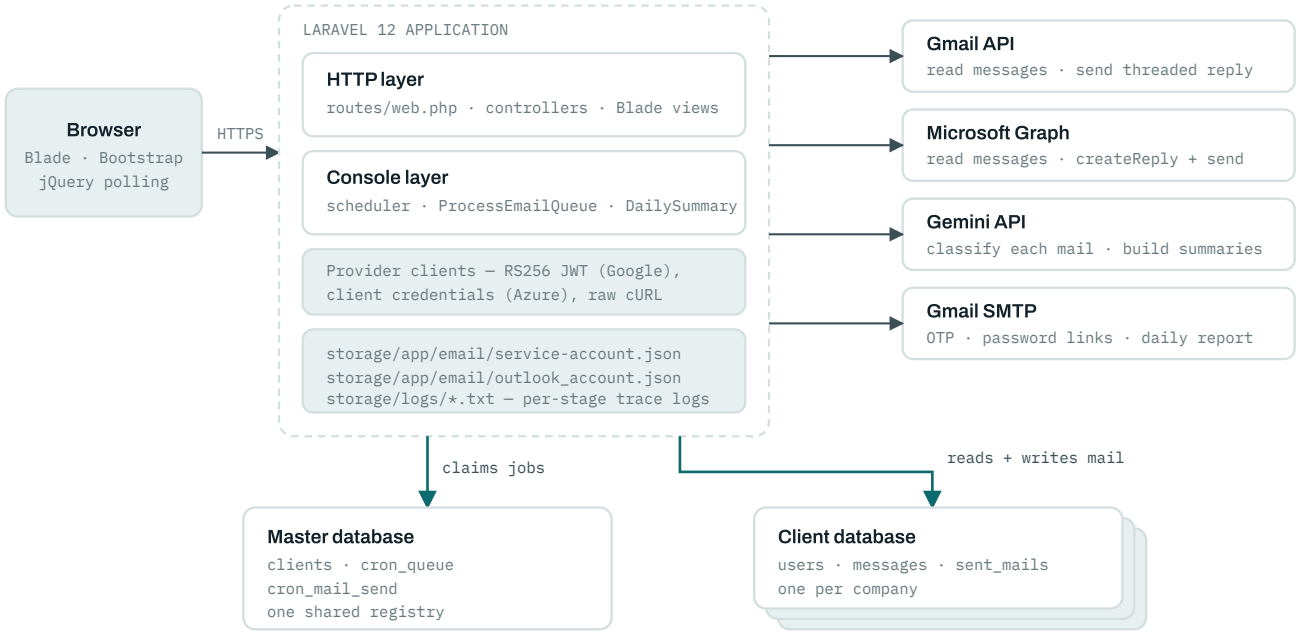

Fig. 4 — The application holds no mail credentials per user: provider access is a single service account (Google) or app registration (Azure) that impersonates each mailbox. Company data is physically separated into one database per client, selected at runtime.

Multi-tenancy and the work queue

The master database is a registry, not a copy of anyone's mail. It holds the client list and two work queues. A seed script fills the queues once a day; the scheduled commands drain them, and for each job they repoint the `clientdb` connection at that client's schema before touching any data.

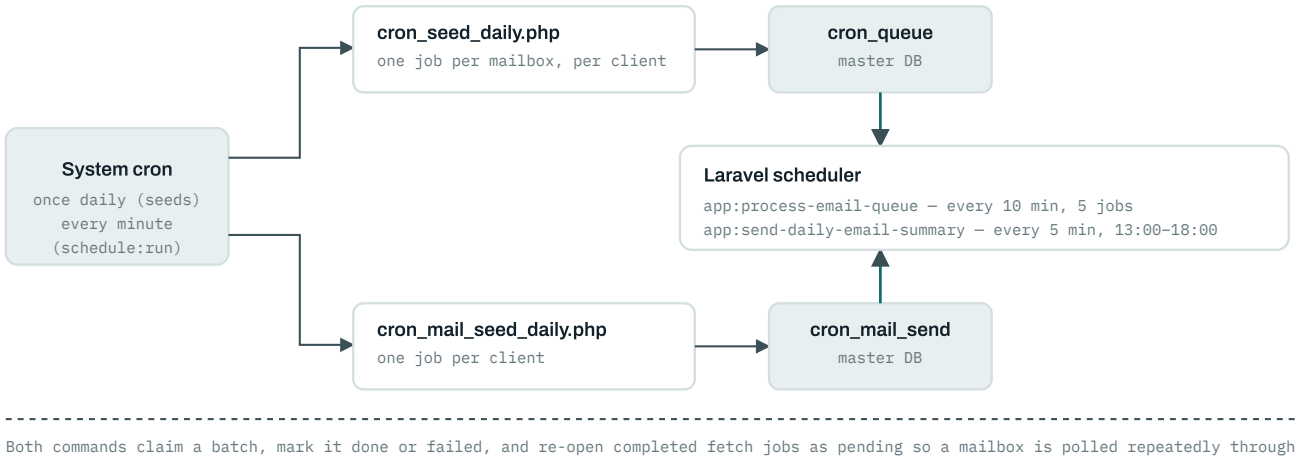

Fig. 5 — Two independent daily queues: one drives mail ingestion per mailbox, the other drives the summary mail per company. Seeding is plain PHP run by the system cron; draining is the Laravel scheduler.

Request flows, end to end

FLOW	PATH THROUGH THE SYSTEM
Sign in	Browser → `POST /login` → password validated → OTP row written → PHPMailer/SMTP → `POST /otp-verify` → session opened
Open a filter	Browser → `GET /mails/{type}/{user?}` → client DB query, de-duplicated by message id → Blade list + toolbar counts

FLOW	PATH THROUGH THE SYSTEM
Fetch new mail (foreground)	Page load → jQuery → `/testing_email_insertion_process` or `/outlook_insertion_process` → provider token → provider API → Gemini → client DB
Fetch new mail (background)	Cron → `schedule:run` → `app:process-email-queue` → master queue → per-client DB → provider API → Gemini → client DB
Reply	`POST /reply-mail/{id}` → service-account token (`gmail.modify`) or Graph `createReply` → provider sends → reply detected on the next ingestion run
Daily report	Cron seed → `cron_mail_send` → `app:send-daily-email-summary` → per-role counts from client DB → SMTP to each admin and head

> **INGESTION HAS TWO ENTRY POINTS**
>
> The same fetch-and-classify logic exists twice: in `MailController` for the browser-triggered path and in `ProcessEmailQueue` for the scheduled path. They have drifted — the command is multi-tenant and queue-aware, the controller is not. Treat the command as the source of truth and the controller path as a manual override.

12

Data model

Everything inside a client database hangs off `users`. There are no Eloquent relationships declared — joins are written explicitly — so the foreign keys below are the contract to respect.

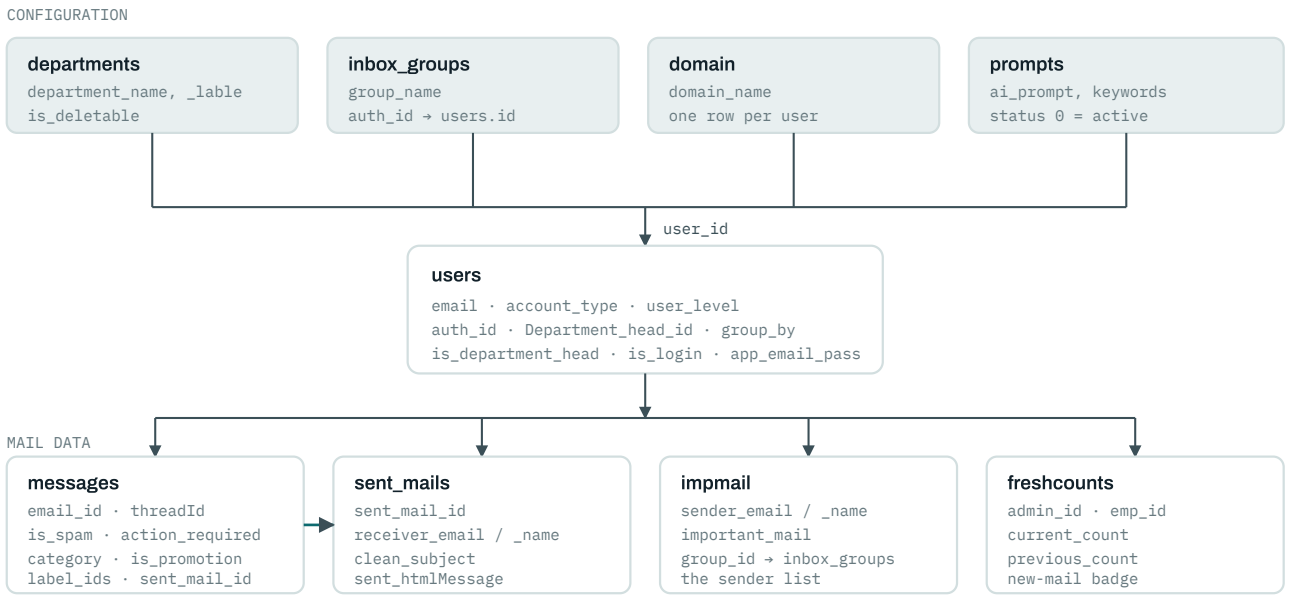

Fig. 6 — `messages.sent_mail_id` is the join to `sent_mails` and the reply flag at once: `-1` means unanswered, any other value points at the message that closed the thread.

Tables in a client database

TABLE	HOLDS	MIGRATION?
`users`	Admins, heads and monitored mailboxes	Yes — extended since
`messages`	One row per fetched message with its classification	Yes — extended since
`sent_mails`	Detected responses, joined back to `messages`	Yes — extended since
`impmail`	The per-mailbox sender list and its important flag	Yes
`domain`	The internal domain used to split senders	Yes

TABLE	HOLDS	MIGRATION?
prompts	Custom AI prompt and keyword label	Yes
promotion_mails	Promotion message ids	Yes
user_emails_id_data	Historic message-id snapshots taken at registration	Yes
departments	Department names and labels	— missing
inbox_groups	Saved sender groups	— missing
freshcounts	New-mail counters per admin	— missing
user_otps	Login codes and their expiry	— missing
project_responses	Referenced by the ingestion command	— missing
password_reset_tokens, sessions, cache, jobs	Laravel framework tables	Yes

Tables in the master database

TABLE	HOLDS
clients	One row per company: company_name, db_name, status, priority
cron_queue	Fetch jobs: client_id, employee_id, status, priority
cron_mail_send	Daily-report jobs: client_id, mail_send, priority

> SCHEMA DRIFT
>
> Several live columns and five whole tables have no migration — `messages.label_ids`, `messages.threadId`, `users.account_type`, `users.user_level` and the master-database tables among them. A fresh `php artisan migrate` will **not** produce a working schema. Provision from a dump of a working database until the missing migrations are written.

13

Reference tables

Routes

METHOD	PATH	PURPOSE
GET	/	Public landing page
GET · POST	/register	Register a mailbox and provider type
GET · POST	/login	Password step; issues the OTP
GET	/otp-verify/{user}	OTP entry form
POST	/otp-verify	Verify code, open the session
POST	/resend-otp	Issue a fresh code
GET · POST	/forgot-password	Request a reset link
GET · POST	/reset-password	Set a new password from a token
GET · POST	/set-password	First password for a new department head
POST	/logout	End the session
GET	/dashboard	Account summary and connection check
GET	/mail-info/{user?}	Mail workspace shell; triggers a fetch
GET	/mails/{type}/{user?}	Filtered mail list
GET	/mails-show/{id}/{is_sent?}/{user?}	Read one message
GET · POST	/reply-mail/{id}/{user?}	Compose and send a Gmail reply

METHOD	PATH	PURPOSE
POST	/reply-OutlookReply/{id}/{user?}	Send an Outlook reply via Graph
GET	/mail/stats?user_id=	JSON counts for the toolbar
GET	/fetchsummary?user_id=	Build the AI summary
GET	/store-new-email	Fetch + classify a Gmail mailbox now
GET	/outlook_insertion_process	Fetch + classify an Outlook mailbox now
GET	/employee	Monitored mailboxes with counts
GET	/employee_process/{id}/{type}	One employee, one signal
GET	/set_email	Sender list — important vs not
GET	/update_set_email?emails[]=…	Save the importance selection
POST	/show_all_mail	All mail from one sender
—	/inbox_groups	Resource routes for sender groups
GET	/departments · /departments/{id}/edit	Department CRUD
GET · POST	/eassign-department	Assign a head and members
GET · POST	/domain · /domain/store	Internal domain setting
POST	/prompt-form	Save AI prompt and keyword
GET	/download-sample	Bulk-upload template
POST	/upload-bulk	Import mailboxes from Excel
POST	/download-excel	Export an import bucket
GET	/checkFreshcount	New-mail counter poll
GET	/up	Framework health check

Artisan commands

COMMAND	DOES	SCHEDULED
app:process-email-queue	Fetch and classify mail for queued mailboxes	Every 10 minutes
app:send-daily-email-summary	Mail the daily report to admins and heads	Every 5 minutes, 13:00–18:00
inspire	Framework sample command, used as a scheduler heartbeat	Every minute

Environment

KEY	USED FOR
APP_KEY	Encryption of stored mailbox secrets — never rotate without re-encrypting
DB_HOST · DB_DATABASE · DB_USERNAME · DB_PASSWORD	This installation's client database
MASTER_DB_HOST · MASTER_DB_DATABASE · MASTER_DB_USERNAME · MASTER_DB_PASSWORD	The shared registry and work queues
GEMINI_API_KEY	Classification and summaries
MAIL_HOST · MAIL_PORT · MAIL_USERNAME · MAIL_PASSWORD · MAIL_FROM_ADDRESS · MAIL_FROM_NAME	Outbound SMTP for OTP, password links, reports
QUEUE_CONNECTION · CACHE_STORE · SESSION_DRIVER	Database-backed by default

CREDENTIAL FILES (NOT IN VERSION CONTROL)

- `storage/app/email/service-account.json` — Google service-account key with domain-wide delegation.
- `storage/app/email/outlook_account.json` — cached Graph access token; recreated automatically when it expires.

LOG FILES

The app writes its own trace logs beside the Laravel log, in `storage/logs/`:

- `MailController_check_load_time_log_cron.txt` — stage-by-stage timings of a fetch run.
- `MailController_log_cron.txt` — ingestion errors and raw AI responses.
- `MessageController_log.txt` and `MessageController_Gemini_response_log.txt` — summary generation.
- `MailController_flask_Api_log.txt` — AI call failures and token usage.
- `Job_fail_log.log` — failed queue jobs.
- `cron_seed_daily.log`, `cron_mail_seed.log` — daily seeding.

14

Running it in production

Requirements

- PHP 8.2+ with `curl`, `openssl`, `mbstring`, `zip` (Excel) and `imap` if you use the legacy app-password path.
- MySQL 8 — one master database plus one database per client company.
- Composer dependencies; Node and Vite only if you rebuild front-end assets.
- A Google Workspace service account with domain-wide delegation, and/or an Azure AD app registration with application-level Mail permissions.
- A Gemini API key, and SMTP credentials for outbound mail.

Install

01 `composer install --no-dev --optimize-autoloader`

02 Copy `.env.example` to `.env`, then fill in both database blocks, `GEMINI_API_KEY` and the mail settings.

03 `php artisan key:generate` — do this once, before any mailbox secret is stored.

04 Load the schema. Because migrations are incomplete, restore from a known-good dump rather than running `migrate` on an empty database.

05 Drop the two credential JSON files into `storage/app/email/`.

06 Register the company in the master `clients` table with this installation's `db_name` and `status = 'active'` — without that row, nothing is ever queued.

07 Point the web root at `public/` and make `storage/` and `bootstrap/cache/` writable.

Scheduling

Three cron entries drive everything. The seeds create the day's work; the Laravel scheduler drains it.

WHEN	RUN	WHY
`* * * * *`	`php /path/artisan schedule:run`	Drives both scheduled commands
`once each morning`	`php public/cron_seed_daily.php`	One fetch job per mailbox, per active client; clears yesterday's queue
`once before 13:00`	`php public/cron_mail_seed_daily.php`	One report job per active client

> **MOVE THE SEED SCRIPTS OUT OF THE WEB ROOT**
>
> Both seed scripts sit in `public/`, which makes them reachable over HTTP, and both carry database credentials inline. Move them outside the document root (or block them in the web server) and read credentials from the environment instead.

Capacity

Per run	5 mailbox jobs
Per mailbox per run	up to 40 messages, listed 200 at a time
Cadence	every 10 minutes → roughly 30 mailboxes an hour
AI calls	one classification call per message, plus retries on 503

Ingestion throughput is bounded by AI rate limits far more than by the database. If a backlog builds, raise the batch size and the per-mailbox cap together, and watch for 429 responses in `MailController_flask_Api_log.txt`.

15

Troubleshooting

SYMPTOM	MOST LIKELY CAUSE	FIX
"Service account JSON file not found"	Missing or unreadable `service-account.json`	Restore the file to `storage/app/email/` and check ownership
"You Are not Company Team member"	Impersonation refused: mailbox outside the delegated domain, or the scope is not authorised	Add the mailbox to the domain, and grant `gmail.readonly` plus `gmail.modify` to the service-account client id
"Token file not found" (Outlook)	`outlook_account.json` missing	Create the file; it is refreshed automatically once a token is obtained
Dashboard says "Please try Later"	The provider connection check failed on page load	Look at the provider credentials first — this is not a password problem
No OTP arrives	SMTP rejected the send	Verify `MAIL_USERNAME` / `MAIL_PASSWORD` and that the sending account allows app passwords
Counts never change	No `clients` row, no seeded jobs, or `schedule:run` is not in cron	Check the master tables, then `cron_seed_daily.log`
Mail stops mid-import	Gemini returned 429 or 503 and the run aborted	Wait for the next tick; if persistent, lower the batch or raise the quota
Everything classified `101`	The model is not returning bare JSON	Inspect `gemini_return_json`; the parser strips code fences only
A reply you sent still shows as pending	Reply detection runs on ingestion, not on send	Wait for the next run — see Fig. 2
Employee list is empty for a head	Nobody has `Department_head_id` set to them	Assign members under **Assign Department**
Job marked failed	Exception during fetch or classify	Read `Job_fail_log.log`; the job is retried on the next seed

16

Findings to address

These came out of reading the code for this document. None of them stop the product working; all of them are worth fixing before it carries more customers. Ordered by severity.

SEVERITY	FINDING	WHERE
critical	**Live secrets are hard-coded in source.** An Azure tenant id, client id and client secret; a Gmail SMTP app password in three files; a Gemini API key in two more; the master-database password in both seed scripts and as a fallback default in the importer. Anyone with repository access has production credentials.	`AuthController · InboxGroupController · ForgotPasswordController · MessageController · ExcelController · p`
critical	**No route middleware.** Not one route is wrapped in `auth`. Controllers check `Auth::check()` individually and several do not — including the fetch endpoints, `/mail/stats`, `/fetchsummary`, `/show_all_mail`, `/mail_with_email_id/{id}` and `/checkFreshcount`. Because mailbox scope comes from a URL parameter, an unauthenticated request with a guessed id can read another company's mail.	`routes/web.php`
critical	**Secrets in the web root.** `cron_seed_daily.php`, `cron_mail_seed_daily.php` and `dialogflow.php` are publicly reachable and contain credentials; the first two also mutate the queues on request.	`public/`
high	**Migrations do not describe the schema.** Five tables and many live columns exist only in production. A clean install from migrations produces a broken app.	`database/migrations/`

SEVERITY	FINDING	WHERE
high	**Level-0 users are unfiltered.** The visibility branches restrict level 2 and level 1 but leave the level-0 branch empty, so that path returns every user. Login blocks level 0 today, which is the only thing preventing exposure.	`EmployeeController` · `MailController`
high	**Two divergent copies of the ingestion pipeline.** `MailController` (about 5,400 lines) and `ProcessEmailQueue` implement the same fetch-and-classify logic with different behaviour; only the command is tenant-aware.	`MailController` · `ProcessEmailQueue`
medium	**Debug behaviour on live paths.** A failed OTP send calls `dd()`, and AI transport errors call `die()` mid-request — both dump to the browser and abort. Several controllers `echo` diagnostics into rendered pages.	`AuthController` · `MailController` · `DashboardController`
medium	**A fixed internal address is CC'd on every OTP mail.** Login codes for every customer are copied to one inbox.	`AuthController`
medium	**Raw SQL interpolates the current user id** in the sender-list subqueries. It happens to be safe because the value comes from the session, but it is one edit away from injection.	`MessageController`
medium	**Bundled PHPMailer loaded by `require` from a hand-placed vendor folder**, in parallel with the Composer-managed copy. Two versions can disagree.	`AuthController` · `InboxGroupController` · `ForgotPasswordController`

SEVERITY	FINDING	WHERE
low	**Backup copies committed alongside live code** — three controller directories, three view directories, and `.bak-design` beside most Blade files. They shadow the real files during review.	`app/Http/` · `resources/views*/`
low	**Test routes exposed and a debug `alert()` on page load** in the mail workspace; the README is still Laravel's default.	`routes/web.php` · `mailinfo.blade.php` · `README.md`
low	**Dead code.** `GmailService` (1,700 lines) is never instantiated; many `if (true) { } else { … }` blocks keep both an old and a new implementation in the file.	`app/Services/GmailService.php`

SUGGESTED ORDER OF WORK

1. Rotate every credential that appears in source, then move all of them to `.env`.
2. Put an `auth` middleware group around every authenticated route, and authorise the mailbox id in the URL against the signed-in user.
3. Move the seed scripts out of `public/`.
4. Write migrations for the five missing tables and the added columns, and verify a clean install.
5. Retire the controller copy of the ingestion pipeline in favour of the command.