PERSONAL PROJECT · CASE STUDY

Gmail Sorting Agent

An inbox agent that decides what every job-search email actually is, labels it, and clears the noise out — built around one hard rule: it is allowed to leave junk in your inbox, but it is never allowed to bury an interview.

`Python` `Gmail API (OAuth2)` `Claude API (Haiku)` `JSON local memory` `85 automated tests` `11 categories · 12 labels` `Git`

01 The problem, and the idea

Applying for jobs at volume fills an inbox with mail you will never read twice. Application receipts, *"your application was viewed"* pings, daily job alerts, platform nudges to complete your profile. Hundreds of them, arriving between the four or five emails that actually matter — an interview time, an assessment link, a request for documents. The obvious automation is a filter that archives anything matching *"thank you for applying"*. That build is wrong in a specific and expensive way: the same sender, the same subject line, and often the same opening paragraph are used for both the receipt and the request.

So the agent has to actually decide. And the decision runs two rules that pull against each other — the engineering is entirely in how they're reconciled.

RULE 1 · BE USEFUL

Get the noise out

An inbox holding 400 unread receipts is an inbox you stop opening. Removing mail is the entire value of the thing — an agent too timid to archive anything is a rounding error with an API key. It has to be willing to act on hundreds of emails at a time.

RULE 2 · BE SAFE

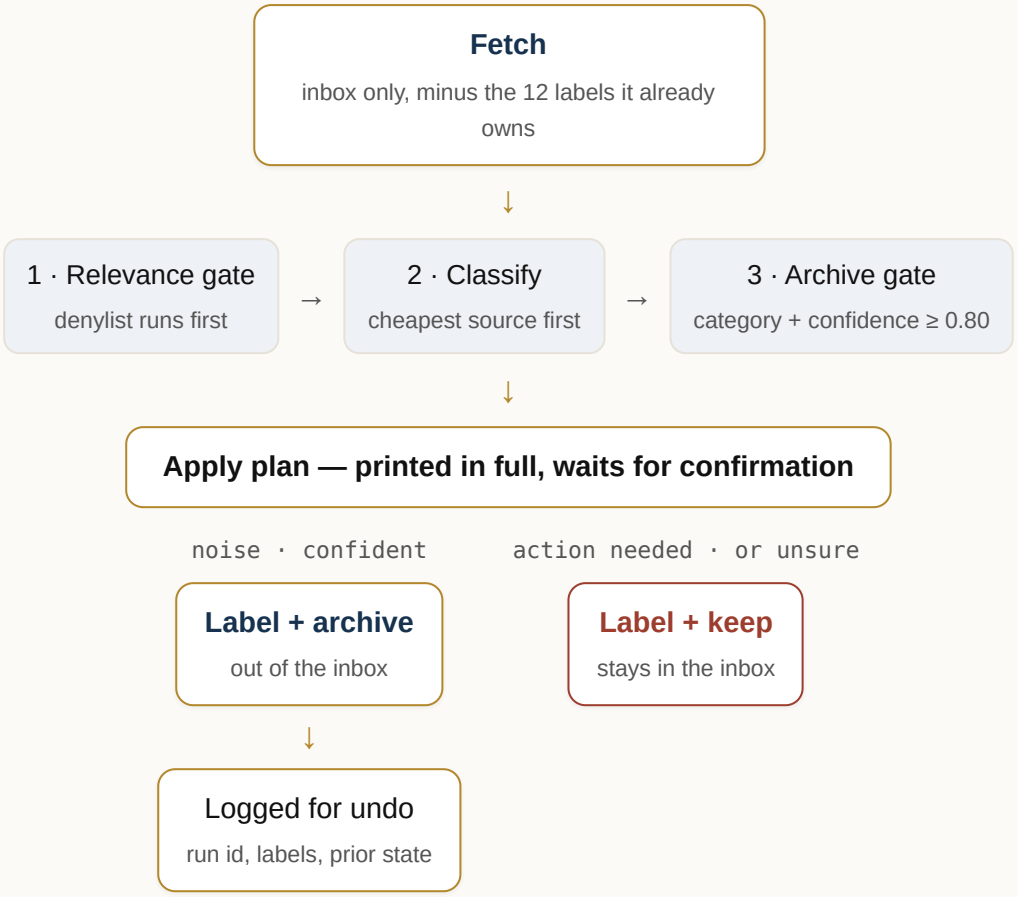
Never bury a task

An archived interview invite is an opportunity that ends silently. This failure has no feedback signal — you don't notice the email you never saw. So false negatives are treated as categorically worse than clutter, and uncertainty always resolves to the inbox.

The agent is allowed to fail at **tidying your inbox**.
It is not allowed to fail at **protecting an opportunity**.

02 How an email becomes a decision

Every email runs through a fixed funnel. Most of the inbox is eliminated at the relevance gate for free and never touched again; only job-search mail reaches the classifier, and only a confident, archivable classification reaches the inbox itself.



Not job-related? Left completely untouched — never labelled, never archived.

TWO GATES The two gates fail in opposite directions, on purpose. The **relevance gate** is deliberately generous — it would rather admit a newsletter than miss a real application, so a wrong answer costs a label. The **archive gate** is deliberately strict — an email leaves the inbox only if its category is archivable *and* confidence clears 0.80, because a wrong answer costs an opportunity. Cheap mistakes are permitted at the top of the funnel precisely so expensive ones can be refused at the bottom.

CATEGORY, NOT JUDGEMENT Eleven categories, split into two lists. Six always stay: *interview invite*, *assessment due*, *recruiter message*, *documents needed*, *offer*, *review manually*. Five may be archived: *applied*, *rejection*, *application viewed*, *job alerts*, *platform noise*. Which list a category sits on is data in a table, not a decision made at runtime — so a rule cannot archive a keep-in-inbox category even when it explicitly declares itself safe to archive. The claim is simply ignored.

03 Classifying an email — cheapest source first

This is the distinctive engineering. Claude is a component inside the agent, not the architecture of it: four local sources are exhausted before a single API call, and the model isn't reached at all unless it was explicitly switched on for that run. One gate sits across all six tiers — including the model's own answer.

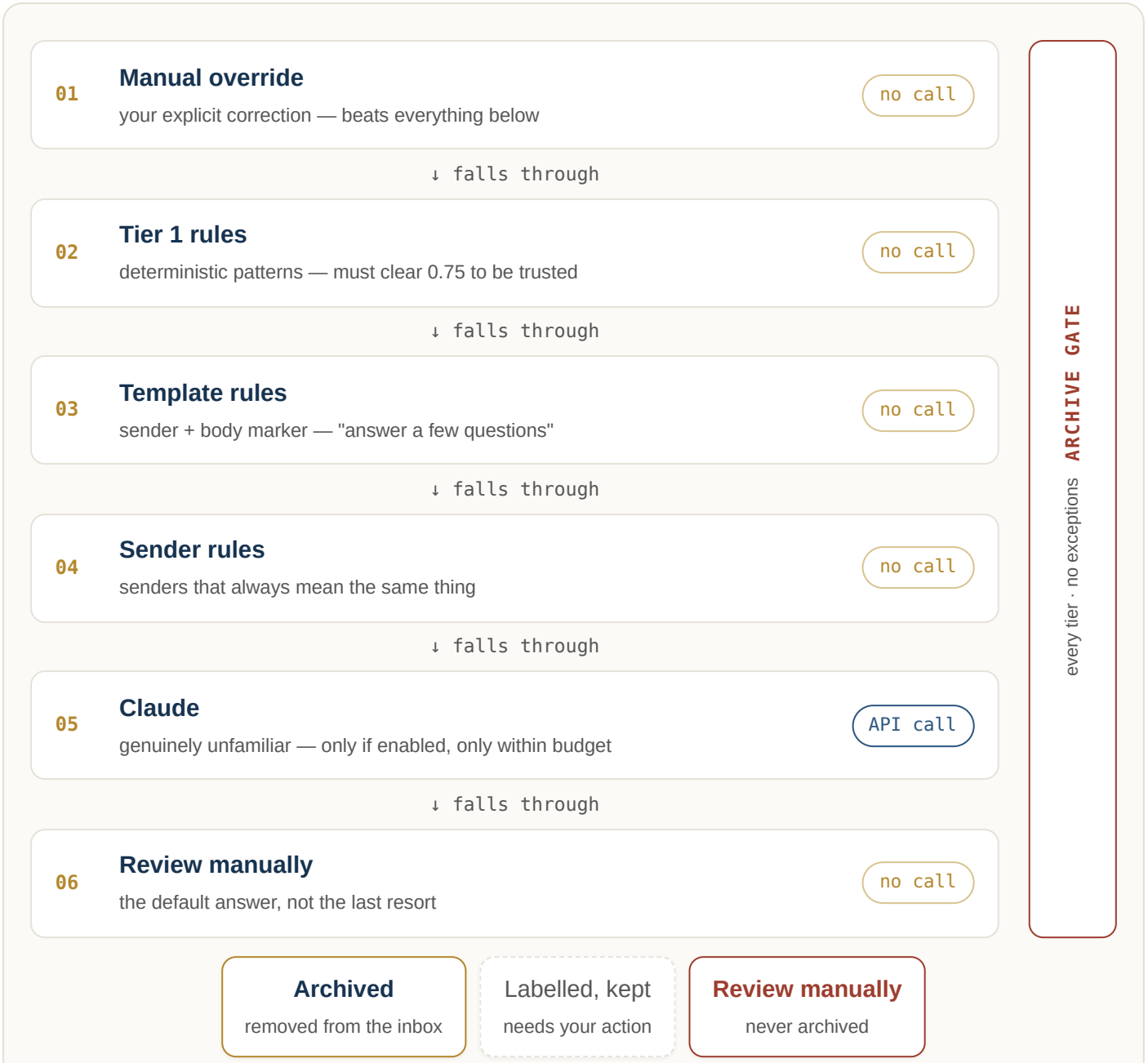


Figure 1 — The six-tier classification cascade. Four free sources are exhausted before a single API call is made, and the gate spans every tier: the model's own answer is not trusted merely because a model produced it.

THE FLOOR Tier 06 is the reason the whole structure holds. In most designs the model *is* the fallback — whatever the rules can't handle, the API answers, and the API always answers something. Here the fallback is **review manually**, and the model is an optional shortcut on the way to it. Turn Claude off entirely and the agent still runs, still labels, still archives, and simply gets more conservative. That property is what makes the API key optional rather than load-bearing.

04 What keeps it safe

Every claim below is a rule in code with a regression test behind it, not an instruction in a prompt. A prompt can be talked out of its principles; a guard clause cannot.

Uncertainty is never archived

Anything it can't place confidently becomes *Review Manually* and stays put. There is no code path from "unsure" to "archived" — the fallback isn't a guess, it's leaving the email alone.

A known sender can't bury a real task

If interview, assessment, document or offer language is present — or an embedded action like *confirm your residency* — every archive-shaped rule is suppressed and the email falls through to review.

It doesn't trust its own model

A Claude answer passes the same gate as everything else: archivable category, confidence ≥ 0.80. A category it doesn't recognise is coerced to *Review Manually* rather than passed through.

It reads the body, not just the subject

"Thank you for your application" that also asks you to confirm your residency is a task, not a receipt. A specific request in the body overrides a reassuring subject; generic boilerplate doesn't.

It cannot delete anything

Archiving removes one label; everything stays in All Mail, fully searchable. The OAuth scope it holds cannot send or delete mail — that isn't policy, it's the permission it was granted.

Every run is reversible

Each run has an ID; every label and archive is logged with the email's prior label state. Undo replays a run backwards and makes the agent forget it ever touched those messages.

It can't spend without being asked

Claude is off by default, needs a flag, and obeys a per-run call cap. Past the cap the remaining emails become *Review Manually* rather than more spend — the budget degrades safety-ward.

It shows its work before acting

A dry run writes no state at all. A real run prints every label and every archive it intends, then waits. Abort and nothing is written — an aborted run is not a remembered run.

05 Under the hood

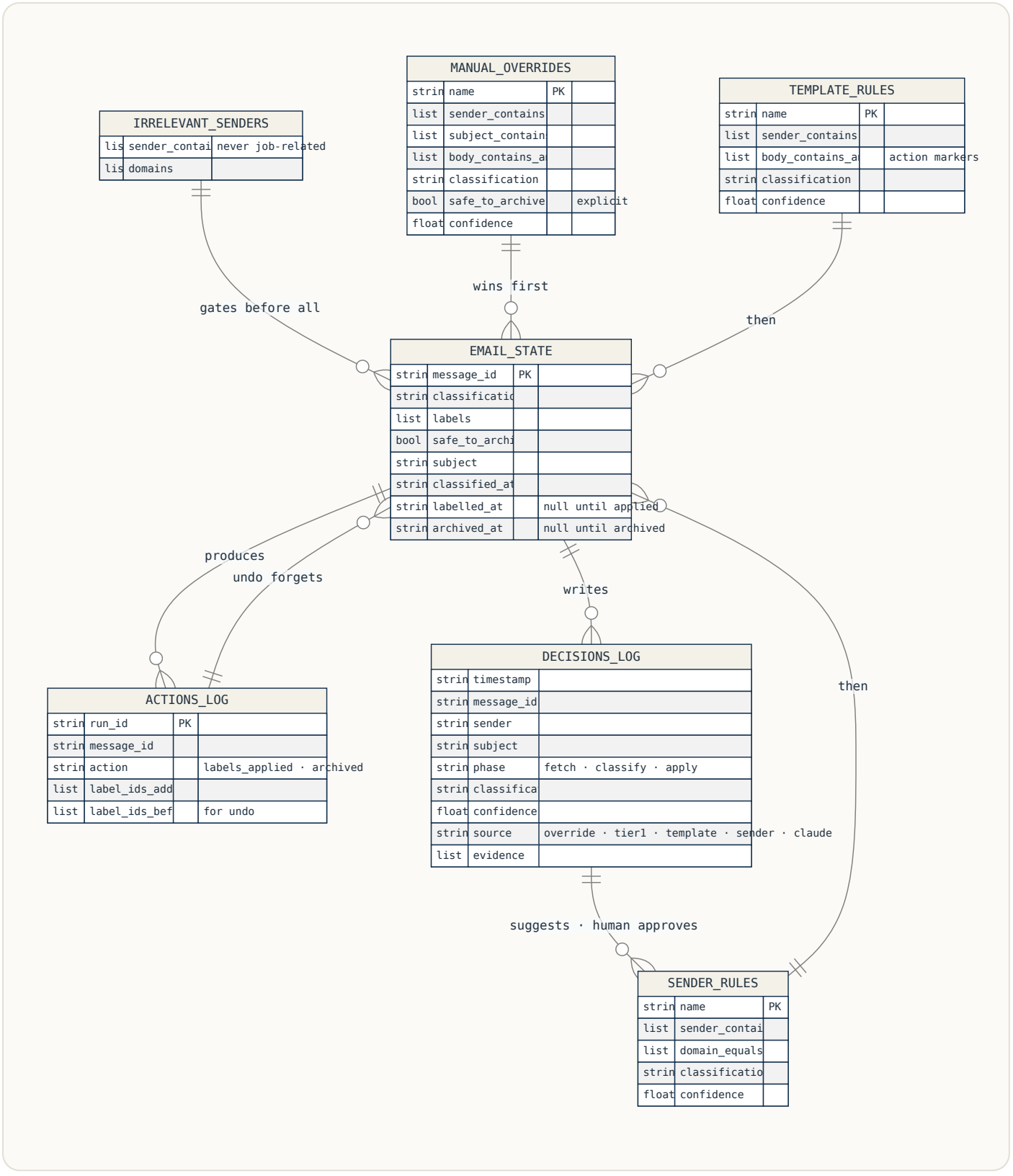
A modular Python CLI over the Gmail API, holding a `gmail.modify` scope — enough to read, label and archive; not enough to send or delete. The components below are the ones carrying a safety or cost decision.

COMPONENT	WHAT IT DOES
is_relevant	The first gate. Denylist, then job-platform domains, sender hints, keywords in subject and body, and link domains. A denylisted sender is skipped entirely — not classified, not labelled, not touched.
_tier1_classify	Deterministic pattern matching across 11 categories, returning a confidence. Below 0.75 the result is treated as a tentative note and the email continues down the cascade rather than being acted on.
LocalRules	Loads and matches the four JSON rule files. A malformed rule is skipped with a warning rather than crashing the run; a rule with no match conditions never fires, so an empty rule can't silently catch every email.
_has_high_value_signal	The suppressor. Returns true on interview, assessment, document, offer or embedded-action language — which disables every archivable rule for that email, no matter how confident the rule was.
_make_rec · _rec_from_rule_match	The archive gate, and the only place safe_to_archive is ever set. Requires an archivable category and confidence ≥ 0.80. Unknown categories are coerced to review_manually; a rule's own archive claim is ignored unless the category allows it.
classify_with_claude	Tier 2. Sends a compact evidence package rather than the raw email — sender, subject, ~1,000 body characters, ranked link metadata — so what leaves the machine is inspectable and minimal.

COMPONENT	WHAT IT DOES
`_score_link · _rank_links`	Ranks extracted links before the model sees them. Action-oriented anchors and button-styled links are boosted; unsubscribe, social and privacy links are pushed down. Top 8 only — the buttons usually reveal the email's real purpose better than its subject.
`EmailState`	Per-message record of classified / labelled / archived as three separate timestamps, written atomically. Dry runs write nothing; a failed Gmail call never records success.
`build_search_query`	Builds the Gmail query, excluding the 12 labels the agent owns so handled mail is never re-fetched. Pure function, unit-tested without touching the API.
`suggest_job_search_rules`	Offline tool. Mines the decision log for senders classified identically enough times and proposes rules — stricter thresholds for archivable ones, and never a rule from uncertainty. Read-only: it prints, a human merges.

06 Data model

There is no database. Memory is four rule files, one state file and two logs on disk, which is the right shape for a single-user agent whose rules must be readable, hand-correctable and diffable in Git. The relationship that matters is the loop at the bottom: **decisions_log** is mined into **sender_rules**, so the agent's own past is what makes its future free.



PK key field **one ──< many** (one log suggests many rules)

JSON document stores, not tables

Figure 2 — Four rule files, one state file, two logs. The bottom edge is the learning loop; the edge above it is undo, running the other way.

WHY JSON A relational store would be the reflex choice and the wrong one here. The rules are edited by hand constantly — a wrongly archived email is fixed by opening a file and adding four lines. Keeping memory as readable documents means every rule change is reviewable in a Git diff, and the safety-critical ones stay legible to the person accountable for them.

THREE STATES An email is **classified**, then **labelled**, then **archived** — three timestamps, not one "processed" flag. The distinction is load-bearing. A flat processed list once marked labelled emails as finished, so the archive pass skipped them and they sat in the inbox wearing a label forever. Splitting the states also fixed the archive cap: emails beyond the limit stay eligible and the next run drains them, without re-classifying anything.

07 Decisions under tension

Four design conflicts where the obvious answer and the safe answer pointed in different directions. These are the decisions the architecture is actually made of.

CONFLICT 01 · ORDERING

The cheap rules should run first. They don't.

The design called for sender and template rules ahead of the deterministic Tier 1 rules — they're free and fast, so check them first. But the same design required that high-value action language always beat a broad rule. Those two requirements contradict each other: a sender rule saying *"this ATS only ever sends receipts"* would fire before anything had read the body and found *"confirm your residency"*.

Resolved: confident Tier 1 stays above the broad rules; template and sender rules fill the gap *below* Tier 1's 0.75 threshold — which is exactly where the API calls were coming from anyway. The cost saving is kept and the guard survives. The brief was treated as a design input, not a specification to implement literally.

CONFLICT 02 · LEARNING

Auto-learning was the headline feature. It was cut.

The plan included a fingerprint cache: hash the email template, store the classification, reuse it forever. It would have driven API calls close to zero — genuinely the most effective single feature available.

Resolved: deferred, and a suggester built instead. A cache keyed on a fingerprint doesn't just learn — it repeats a single mistake indefinitely, silently, with nothing to review. The suggester mines the same history and proposes the same rules, but a human merges them. Slower to learn; structurally unable to learn something wrong. The same reasoning keeps the relevance denylist hand-edited: auto-denylisting a sender that occasionally carries real mail is the one failure with no feedback signal at all.

CONFLICT 03 · UNDO

Undo forgets rather than corrects.

When an archive is reversed, the obvious repair is to clear the *archived_at* timestamp. That's wrong, and quietly so: the email is then labelled, marked safe, and un-archived — precisely the state the drain pass looks for. The next run would re-archive the email the user just rescued.

Resolved: undo removes the message from state entirely — the agent treats it as untouched. Permanent *never archive this* is expressed as a manual override, reusing the mechanism that already exists rather than inventing a second one. Undo only forgets messages whose reversal actually succeeded; a failed reversal keeps its entry, because state should describe reality rather than intent.

CONFLICT 04 · COST

The spend was already trivial. The architecture changed anyway.

At the point the redesign was proposed, the model was handling about 18% of mail at an estimated thirteen cents. There was no cost problem to solve, and a reasonable reading was to leave it alone.

Resolved: rebuilt regardless, because the target was never the average bill — it was the absence of surprise. A model that answers by default is a component that can spend without being asked and answer without being sure. Making it opt-in and capped changed what the agent *is*: the local rules became the system and the model became a shortcut. The most recent production run made zero calls — not because Claude was unavailable, but because nothing that day was unfamiliar enough to need it.

08 What it got wrong

Every guardrail in section 04 exists because something failed first. These are the real ones, found in a real inbox — including the one still left unfixed on purpose.

BUG 01 · FIXED

The receipt that was actually a task

An email titled *"Thank you for your application!"* from a recruiter's system was classified *applied* and archived. The body asked the candidate to confirm their residency status and desired work location. A real task, buried under a reassuring subject line.

Cause and fix: the prompt instructed the model to weigh the subject *"REGARDLESS of boilerplate in the body"* — a rule written to stop generic legal footers hijacking classifications, which also taught it to ignore genuine requests. Rewritten to separate boilerplate (*"references may be checked"* — ignore) from specific action requests (*"confirm your residency"* — classify by the action). The case is now pinned by a manual override, verified against the original email.

BUG 02 · FIXED

A person, mistaken for a platform

"Ramdas just messaged you" — a direct message from a real human — was classified *platform noise* and archived.

Cause and fix: the sender address was a *noreply* LinkedIn digest address, and a noreply sender triggered the platform-noise fallback. The agent had judged the envelope rather than the message. Direct-message subjects are now detected before the noreply guard runs, and resolve to *review manually* — a stranger's message is not something the agent should be confident about either way.

BUG 03 · FIXED

A gate that wasn't a gate

The archive gate requires confidence ≥ 0.80. The model occasionally returned confidence as `85` rather than `0.85` — and 85 clears 0.80 trivially. For those responses the gate was open, and it looked closed.

Cause and fix: found by reading the code rather than by anything going wrong, which is the uncomfortable part — the failure was silent and would have stayed silent. Values are now normalised from the 0–100 scale, non-numeric responses default to 0.5, and everything is clamped. A guard that trusts its input isn't a guard.

BUG 04 · FIXED

Marketing that spoke fluent job search

Product marketing from an AI company was pulled into the job-search module and archived as platform noise. It matched the relevance filter because its copy contains the word *"application"* — in the software sense.

Cause and fix: the filter was right by its own logic and wrong in fact. Narrowing the keyword was the tempting fix and the wrong one — *application* is the single most load-bearing word in a job-search inbox, and weakening it to catch one newsletter would risk missing real mail. Fixed with a sender denylist instead: a targeted exception rather than a blunted rule.

BUG 05 · OPEN, DELIBERATELY

The one still left wrong

A LinkedIn industry-news email is still archived as platform noise every time it arrives. The fix is one line in the denylist and takes ten seconds.

Why it's still there: that same sender address can carry real recruiter messages. Denylisting it trades a visible piece of clutter for an invisible missed opportunity — which is exactly the trade Rule 2 exists to refuse. The bug is cosmetic; the fix would be structural. It stays.

09 Outcomes

915	700	85	128
EMAILS LABELLED	EMAILS ARCHIVED	AUTOMATED TESTS	MODEL CALLS, ALL TIME

0
MODEL CALLS, LAST RUN

Across two production runs the agent has labelled 915 emails and archived 700 of them. The classification spread from the most recent run is the shape you'd want: 218 application receipts and 49 job alerts sorted for archiving, against 4 interview invites, 3 assessments and 1 offer labelled and left exactly where they were.

The economics are the point. The 90-day reprocess that did the bulk of the work reached the model 128 times across 727 emails — about 18%, with the local rules absorbing the other 82% — at an estimated $0.13. The more useful number is the most recent run: 1,000 emails fetched, 424 classified, 197 labelled, 200 archived, and **zero model calls**. Not because the model was unavailable, but because nothing in that day's mail was unfamiliar enough to need it. That rate falls further as rules accumulate, which is the whole design: every sender recognised once is a sender that never needs the API again.

The safety property worth defending is structural rather than statistical. No email classified *interview invite*, *assessment due*, *documents needed* or *offer* can be archived — not because the model is reliable, but because the archive gate is a lookup against a category table and those categories aren't on the archivable list. The claim is enforced by the type of the thing, not the confidence of the answer.

> **WHAT THIS DOESN'T PROVE** The gate guarantees that an email **classified** as an interview invite is never archived. It cannot guarantee an interview invite is never **classified as something else** — that's a classification problem, and no gate can close it. It's why the high-value guard reads the body for action language before any archive-shaped rule may fire, why uncertainty resolves to the inbox, and why every run is reversible. The architecture assumes the classifier will eventually be wrong.

SCOPE Two windows, deliberately not blended. The **915 labelled / 700 archived** totals span both production runs. The **128 model calls** belong only to the 90-day reprocess that did the bulk of the classification; the most recent run added 197 labels and 200 archives at zero calls. Averaging calls across lifetime emails would flatter the rate, so the denominators are reported separately. The **$0.13** is an estimate from call count at published Haiku rates, not measured telemetry — the agent counts calls, not dollars, and the figure is presented as the estimate it is. The **85 tests** are the current suite; the last full in-repo run confirmed 72 passing, before the final three test files were added.

Python · Gmail API · OAuth2 · LLM application & prompt design · Decision-cascade architecture · Cost-aware API usage · Rules-vs-model trade-off design · Automated regression testing · Structured logging & audit trails · Reversible operations design · Data modelling (document stores) · Git

---

Personal project by Mason Tatafu · figures taken from the agent's own logs (`decisions.jsonl`, `actions.jsonl`, `email_state.json`) and a run of its live test suite. The agent runs locally on Mason's own inbox under a `gmail.modify` scope that cannot send or delete mail; the API key and OAuth token are kept out of source control.