ChatGPT

Feasibility Study: Tabulensis Local Dashboard Integrations

Executive Answer

Service	What You Can Read (via API)	What You *Can't* Read / Limits	Best Integration Mechanism	Auth & Scopes	Notes
Fastmail	Full mailbox data via JMAP: folder list with unread counts, email metadata (to/from/subject, timestamps), even message content if needed [1] [2]. IMAP/POP also available [3].	No native webhook push (must poll or use IMAP IDLE). No built-in analytics or aggregate metrics beyond mailbox data. API calls are account-specific (no org-wide logs). Rate limits in place for abuse [4].	Poll JMAP periodically (e.g. every few minutes) for unread count and new messages. (Optionally, maintain an IMAP IDLE connection for near-real-time new mail notifications.)	API token (Bearer) with JMAP mail scope (or app password for IMAP) [2] [5]. OAuth available for third-party apps with fine scopes (Mail, Contacts, etc.) [5], but not needed for a single-user internal tool.	JMAP is efficient over HT support searchin partial f Use min scope (r read-on Ensure handle persona carefully (subject senders limits ar generou normal

Service	What You Can Read (via API)	What You *Can't* Read / Limits	Best Integration Mechanism	Auth & Scopes	Notes
Cloudflare	Zone analytics (traffic requests, bandwidth, visitors) [6]; security metrics (e.g. threats blocked) [6]; firewall/WAF events (counts, details via GraphQL) [7]; DNS records and status; Workers metrics (requests, errors) via GraphQL [8]; Pages deployments status via REST.	Detailed per-request logs not directly available in real-time (Logpush to storage requires higher-tier plan). Free plan analytics have shorter retention (24h–72h) and possible sampling. Some security event details require at least Pro plan. No native webhook for analytics/ events (must poll).	Polling via APIs: use GraphQL API for aggregated metrics and events (e.g. query last 24h of requests, error counts) [9]. Use REST for configuration (DNS, etc.) and Pages deployment status. (Logpush can be set up for detailed logs if needed, but not local.)	API Token (Bearer) with least privilege: e.g. "Zone Analytics:Read" and "Zone:Read" for DNS, or "Pages:Read" for Pages API [10] [11]. Avoid global API key. Headers: `Authorization: Bearer <TOKEN>` [12].	GraphQ flexible (one en for all a [13]. Gl rate lim request [14] (Gr 320/5m well abo needs. I GraphQ unavail lower p limited zone sta any) or conside upgradi Pages A provide deployn info (lat build st success [15].

Service	What You Can Read (via API)	What You *Can't* Read / Limits	Best Integration Mechanism	Auth & Scopes	Notes
Resend	Outbound email activity: list of sent emails with status and timestamps (each email's latest event: delivered, bounced, opened, etc.) [16] ; Delivery events (delivered, bounced, complaints, etc.) via webhooks [17] [18] ; Sending domains and verification status (SPF/DKIM records, enabled/disabled) [19] ; Suppression events (when an address is auto-suppressed after bounce) [20] .	No single "suppression list" API endpoint (suppressed addresses are handled internally; must infer from events or bounce details). No built-in aggregate counters (e.g. total sent this month) – requires computing from email list. Content of sent emails is available via API retrieval, but attachments are separate endpoints [21] [22] . Rate limit ~2 requests/sec [23] .	Webhooks for real-time events (bounce, complaint, delivered) – Resend can POST to a local endpoint on each event. If a direct webhook endpoint isn't feasible locally, poll the "List sent emails" API periodically and check `last_event` for new bounces [16] . Use local caching to avoid re-fetching unchanged data.	API Key (Bearer `re_xxx`) with **full_access** needed for read endpoints [24] (the "send_only" key cannot retrieve data). Include `Authorization: Bearer <API_KEY>` in headers [25] . Secure the key (it grants email read/send for your account).	All desir is obtai 100% co of requ signals events (from an suppres endpoir suppres events f gap). Us webhoo accurac bounce complai (automa retry fo failure polling, under r (2 r/s) one req per min

Service	What You Can Read (via API)	What You *Can't* Read / Limits	Best Integration Mechanism	Auth & Scopes	Notes
Stripe	Nearly everything: customer & subscription data (to derive MRR/ ARR, active counts), invoice/ payment status (including failures and delinquent payments), charges and payouts, disputes, etc. Stripe's REST API exposes subscriptions, invoices, balance (pending payout amounts), disputes, events, etc. in detail. Webhooks available for all events (invoice paid/ failed, subscription created, charge disputed, payout initiated, etc.).	No direct "MRR" or aggregate revenue endpoint – must compute from current subscriptions or past invoices. Some derived metrics (ARR, growth rates) require client-side calculation or Stripe Sigma (paid feature). Otherwise, no meaningful data is off-limits. API rate limits are high (100 req/s live) – polling large datasets can be done but is not recommended [27].	**Webhooks strongly recommended** for real-time feed of events (e.g. `invoice.payment_failed`, `customer.subscription.created`, `charge.dispute.created`, etc.) to update dashboard immediately. Supplement with periodic polling for summaries (e.g. daily total revenue). If webhooks aren't used, periodically poll key endpoints (list subscriptions, list disputes, etc.), but beware of rate limits and data staleness [27].	Secret API Key (or restricted key) used with Bearer auth. **Use a restricted key** scoped read-only to necessary objects (Customers, Subscriptions, Invoices, Payments, etc.) for least privilege. Include as `Authorization: Bearer sk_live_***`. Webhook signing secret for verifying Stripe webhooks.	All requ info is obtaina API (100 coverag test mo develop Stripe's compre plan tie limit da access. Maintai state fo comput metrics store subscrip info to comput without API call will sen for all re change leverag these is efficien polling

Fastmail (JMAP/IMAP Email API)

Fastmail supports multiple open mail protocols, primarily **JMAP** (JSON Meta Application Protocol) for modern API access, and IMAP/POP for legacy access [1]. For a local dashboard, JMAP is ideal due to its efficiency and JSON structure. Key aspects of Fastmail's API surface:

- **Available APIs:** Full **JMAP** endpoint (`https://api.fastmail.com/jmap/session` for discovery) to fetch mailboxes, emails, and perform searches [2]. IMAP is also available (and could be used with an IMAP library for real-time new mail via IDLE), but not needed if using JMAP. Fastmail has no proprietary REST API beyond these standard protocols – the emphasis is on open standards (JMAP for mail, CardDAV/CalDAV for contacts/calendar) [1].

- **Data you can read:** Essentially *all email metadata and content in your account*. Via JMAP Mail API (per RFC 8621), one can list mailboxes (with unread counts), query messages (e.g. "all unread in Inbox"), fetch message metadata (subject, sender, date) and even body snippets. For the dashboard, the primary data would be:

- **Unread message count:** JMAP `Mailbox/get` returns each mailbox's total and unread counts (Inbox being of interest).
- **Emails by age/SLA:** JMAP `Email/query` can filter or sort emails by date; the client can retrieve timestamps of unread emails to categorize them (e.g. how many unread >24h, >7d old, etc.).
- **Top subject lines or threads:** Using JMAP `Email/get` on a small subset (e.g. the oldest unread, or most recent few) can retrieve subjects. (Subjects and senders are accessible without pulling full bodies by using the `properties` filter in JMAP.)

Fastmail also provides account info like storage quota usage via JMAP, but operational metrics (like "emails per day") are not precomputed – you would derive those if needed by counting messages.

- **Data not available / limitations:** Fastmail does not offer an event streaming or webhook mechanism for new mail; the closest is the IDLE command in IMAP (keeping a socket open) or JMAP's polling for changes. There is no admin API for login logs or similar (beyond what's visible in the UI). All data is per-account (no multi-user data since Tabulensis is a single-user context). Plan-wise, any paid Fastmail plan supports these APIs; no known restrictions on API usage except reasonable rate limiting. The API token must be generated in the account settings (for personal use) – free trial accounts can create these as well. **Rate limits:** Fastmail imposes some rate limits to prevent abuse, though exact numbers aren't public. Excessive requests result in a JMAP `rateLimit` error [4]. In practice, a few requests per minute is fine.

- **Authentication & Scopes:** For personal integrations, generate an **API token** from the Fastmail settings (under "Privacy & Security → Integrations") and use it as a Bearer token in the `Authorization` header [2]. This token is effectively scoped to your account's data. (If using OAuth for a third-party app, Fastmail supports scopes like `urn:ietf:params:jmap:mail` for mail read/write [5], but in this case the API token already grants mail access.) Alternatively, an **App Password** can be used for IMAP/SMTP access [28], but JMAP+token is recommended for our use (it can be made read-only by simply not using any write methods). The API token should be treated like a password – keep it in a secure config.

- **Polling vs. push:** Because Fastmail has no outgoing webhook for new emails, the dashboard will need to **poll**. Feasibility is good: for example, polling the JMAP API every 5 minutes to get the unread count and the oldest unread email's date is low-impact. JMAP's design allows a single request to fetch multiple data points (using "methodCalls" batch in one HTTP round-trip). This means we can get Inbox mailbox status and maybe query for specific emails in one go. The data can be filtered server-side (e.g. JMAP can query only emails in Inbox that are unread and older than X date). If more immediacy is needed, running an IMAP IDLE connection in a background thread could trigger the dashboard to refresh when new mail arrives – but this adds complexity and is likely unnecessary for an internal tool.

- **Percentage of desired info obtainable:** Essentially 100%. **Unread count** – yes (directly from Mailbox state). **Age buckets** – yes (compute from message dates pulled via API). **Top subjects** – yes (fetch a few messages' `subject` and `receivedAt`). No meaningful gap exists; it's all your data. The only minor challenge is the lack of instant push – mitigated by polling. Given a single-user scale, even pulling the full list of unread message headers is quick (and JMAP responses are JSON, easy to parse).

In summary, Fastmail integration is very feasible: use JMAP with an API token to fetch email stats. Keep the token secure and limit scope to mail. The data retrieved (subject lines, counts) should be handled as potentially sensitive (don't log full content). Fastmail's API is stable and built for exactly this kind of client-side integration [1].

Cloudflare (APIs, GraphQL, Logpush)

Cloudflare offers a rich set of APIs for observability and configuration. For a single-operator dashboard focusing on **website status and performance**, the key surfaces are:

- **Available APIs:** Cloudflare's **REST API (v4)** covers account and zone management (DNS records, settings, firewall rules, etc.), and **GraphQL Analytics API** covers detailed traffic and security analytics [6]. Notably:
- *Zone Analytics (REST)* – provides basic aggregates per zone (this older endpoint is now deprecated in favor of GraphQL [29]).
- *GraphQL Analytics* – a unified endpoint (`/client/v4/graphql`) to query metrics for HTTP requests, firewall events, etc., with great flexibility [13]. This is the primary way to get traffic stats (requests, bandwidth), attack counts, and even Worker invocation metrics.
- *Logpush/Logpull* – for enterprise plans, raw logs can be pushed to external storage. This is likely overkill for our use (and not available on free plan).
- *Cloudflare Pages & Workers API* – Cloudflare provides REST endpoints to manage **Pages projects** (deployments listing, statuses) and **Workers** (to upload scripts, etc.). The Pages API (under the main API v4) allows listing deployments and their statuses [15] [30]. There isn't a direct "deploy webhook," but you could poll the deployment status.

- *No native webhooks:* Cloudflare doesn't send push notifications of events to user-defined URLs (except IP/blocklist notifications to email). So integration is pull-based or via external logging.

- **Data accessible to read:** practically all the "operator-facing" info:

- **Traffic metrics:** Total requests, unique visitors, bandwidth, cache hit ratios, etc., over selectable time windows. These are available via GraphQL queries on the `httpRequests1m` (or adaptive) dataset [6]. For example, one can query sum of requests in the past 24h, or a timeseries for trend. (On free plans, the data is typically last 24h with 5 min resolution.)
- **Performance/Origin metrics:** e.g. error rates (HTTP 5xx counts), latency (if needed) – also via GraphQL or zone analytics.
- **Worker errors:** Cloudflare's analytics can break down Worker script outcomes. The GraphQL **Workers** dataset allows querying by script name for invocations and failures [8]. This means we can programmatically get the number of exceptions or failed requests in Workers (e.g. if our site uses Cloudflare Workers). Another approach is using the Workers API to stream logs (via `wrangler tail`) for debugging, but for a dashboard count, GraphQL is simpler.
- **Deploy status:** If using **Cloudflare Pages** for the website, the REST API can list the latest deployment for the project, including its `status` (e.g. "success" or "failure") [15]. This can tell the operator if the last build of the site failed. For Workers, deployments happen via API/CLI, and a successful publish doesn't have a "status" to poll – it either goes through or returns an error. So for Workers, "deploy status" might be interpreted as "is the latest code version running" (which, if publish succeeded, it is). If needed, one could store a version tag and update it on manual deployments.
- **DNS health:** Cloudflare's API can show the zone status (e.g. "active" if DNS is properly setup on Cloudflare, vs "pending" if not). It also can list DNS records – the dashboard could verify important records (e.g. MX, DKIM for email) are present. Additionally, Cloudflare offers an **analytics ping for DNS** (resolution times), though that's less crucial. If the domain is registered through Cloudflare, expiration info might not be in the API (the dashboard could remind via other means).

- **WAF/Security events:** The number of threats blocked (e.g. firewall rules, DDoS mitigations) is available. GraphQL can query the `firewallEventsAdaptive` table to count events (e.g. how many requests were challenged/blocked in the past day) [7]. For detail, GraphQL can fetch recent event samples (IP, rule triggered, etc.), but for a high-level dashboard, an aggregate count or a list of the latest few events (with time and action) suffices. *Note:* On the Free plan, the WAF is limited (only custom rules and Super Bot Fight Mode). However, the dashboard can still show e.g. number of blocked requests (even free plan users see "x threats blocked" in their dashboard, which we can replicate via API).

- **Data not available or plan-dependent:**

- **Log data:** Full request logs (each request's details) are only available via **Logpush** on Enterprise (and partially on Business for some products). Our use-case doesn't require raw logs; aggregated data covers the "signals." If deeper analysis needed (e.g. examining specific attack IPs), the API might not provide *history* beyond the retention window, but that's beyond MVP.
- **Retention and granularity:** Free and Pro plans typically retain analytics for the last 24 hours (48 hours on Pro) at 1-min granularity, and 7 days at 1-hour granularity. Enterprise gets 30 days. The GraphQL API will return an error if you query outside your plan's retention window. So if the dashboard wants monthly trends, it would need to store data locally beyond the API's window. Initially, focusing on real-time (today's metrics) avoids this issue.
- **Certain features:** Some data only exists if you use that Cloudflare feature. For example, **Health Check** results (for origin monitoring) are available via API if configured, but if Tabulensis isn't using

Cloudflare's health checks, then nothing to fetch. Similarly, if no Workers or no WAF rules, those data sets might be empty. There are no "billing" or cost usage via API except retrieving your plan info.

- Cloudflare does have a notion of account-wide analytics (e.g. for all zones) but those require at least a Pro plan in some cases. Given Tabulensis likely has one primary zone, zone-level queries are fine.

- **Auth model:** Use an **API Token** with restricted permissions – this is the recommended approach (as opposed to using the global API key) [31] . We will create a token in Cloudflare's dashboard with only the needed scopes:

- For analytics: "Zone Analytics: Read" (for the specific zone) [31] . This covers GraphQL queries for that zone's traffic and firewall data. In fact, Cloudflare provides a pre-made template for "Analytics API Access" which can be restricted to a zone.
- For DNS or other info: "Zone: Read" to list DNS records or get zone settings.
- For Pages: "Cloudflare Pages: Read" if using the Pages API (that scope allows reading deployments) [10] .
- For Workers metrics: might need "Account Analytics: Read" since Worker metrics are account-level. Alternatively, a separate token with account-level read can be used for that GraphQL query (GraphQL queries for Workers require the `accountTag` and appropriate permission).

The token is used as a Bearer token: e.g. `Authorization: Bearer <token>` in requests [12] . We should avoid granting write permissions or all zones if not necessary. Cloudflare tokens can be scoped to just one zone by ID for zone-level access.

- **Rate limits and constraints:** Cloudflare's API has a generous **global limit of 1,200 requests per 5 minutes** per user [14] . Our dashboard will make only a few calls per cycle, far below this. The GraphQL API has its own limit measured in "query cost" – roughly max 320 queries per 5 min by default [9] . Each query's cost depends on complexity (fields fetched, time range, etc.). Our queries (e.g. a 24h summary) are low-cost. We should still batch metrics into as few queries as possible (GraphQL can return multiple metrics in one response). Pagination may apply if we list resources (DNS records, or get >100 events, etc.), but for summary stats we usually request aggregated counts directly (no pagination needed). One note: the GraphQL API is accessible on all plan levels for basic data. In case of permission issues (some users have reported needing at least Pro for certain GraphQL queries), we would verify the token's scopes and possibly limit to simpler queries. The Reddit thread suggests GraphQL works on free when properly scoped [32] . So likely no issue.

- **Integration approach (polling):** Since Cloudflare won't push data to us, the dashboard should poll at an appropriate interval. For example:

- Poll the GraphQL API for traffic and security metrics every 5 minutes (or 1 minute if very time-sensitive, but 5 is usually plenty for an operator dashboard). This gives near-real-time insight without hitting limits.
- Poll the Pages deployment status endpoint maybe every 15 or 30 seconds during a deploy, or just regularly if expecting frequent deploys. (Alternatively, Cloudflare Pages provides Deploy Hooks – a URL it can call on deploy events – but those are meant for CI integration, not to send status to arbitrary endpoints. So we'll stick with polling the deployments API for latest status.)
- DNS changes or certificate status likely don't change often, so those could be polled daily or on demand. The dashboard might just load them once on start.

- Worker metrics can be polled via GraphQL similar to zone metrics (e.g. daily or hourly).

- **Feasibility of obtaining desired signals:** Essentially **100%** of the listed Cloudflare signals can be obtained:

- *Website traffic:* absolutely, via GraphQL (e.g. sum of requests in last day, or active connection count, etc.) [6].
- *Worker errors:* yes, via GraphQL `scriptName... status` metrics (we can query count of `status:"error"` vs `status:"success"` for the worker) – Cloudflare's docs show querying Workers analytics over time [8].
- *Deploy status:* yes for Pages (via REST GET deployments which returns status of latest deployment stage) [15]. For Workers, if relevant, we consider a deploy successful if no error event; in any case, the integration can focus on Pages if that's how Tabulensis deploys web content.
- *DNS health:* indirectly yes – we can check that zone is "active" (API returns a status field). Also we could resolve a known record via a DNS query from the server itself as an extra check (not Cloudflare API, but e.g. a local `dig` to verify DNS is correct).

- *WAF/security events:* yes, via GraphQL or the firewall events API. For example, we can query `firewallEventsAdaptive` for count of events in last 24h [7], or retrieve the last N events. Note that to get detailed firewall logs, Cloudflare might require Enterprise, but just counting events or listing ones in the recent period is available to all plans via the analytics API [32].

- **Security considerations:** The API token should be stored securely (it grants read access to potentially sensitive data like DNS config). Ensure the dashboard doesn't expose it. Also, be mindful not to inadvertently expose IP addresses or other sensitive event details if the dashboard is shown publicly (not likely, since it's internal). Cloudflare data is less personal than email or Stripe, but still, things like IP of an attacker or path URLs might appear if we pull event samples. For MVP, maybe just aggregate counts to avoid sensitive detail.

Overall, Cloudflare integration is high ROI: after initial setup of the token and queries, it provides a comprehensive view of site health. We'll need to write some GraphQL queries (Cloudflare provides examples for firewall events [7] and other metrics) and parse JSON results. No significant maintenance expected, as these APIs are stable. If any ambiguity arises (e.g. GraphQL query errors), testing in Cloudflare's GraphiQL or with curl as per docs will help.

Resend (Email Sending API & Webhooks)

Resend is an email-sending service (transactional email API), so the dashboard will monitor outgoing email health – things like sent volume and delivery problems. Resend's integration points:

- **Available APIs:** Resend provides a **RESTful API** with endpoints for sending email and for retrieving information about sent emails, domains, etc. Key parts:
- **Email sending/records:** We have `GET /emails` ("List Sent Emails") which returns a list of sent email objects [16]. Each email object includes metadata (to, from, subject, timestamp) and a `last_event` field representing the latest delivery status (e.g. "delivered", "bounced", "opened") [16]. There is also `GET /emails/{id}` to retrieve details of a specific send (which can include error messages or SMTP response if bounced).

- **Domains:** `GET /domains` to list sending domains and verification status, and `GET /domains/{id}` for detail. This shows whether DNS (SPF/DKIM) is set up (fields like `status` and the required DNS records with statuses) [19].
- **Webhooks:** Resend allows configuring webhooks (via API or dashboard) for various **event types** – e.g. email delivered, bounced, complained (spam report), opened, clicked, etc. [17] [18]. This means Resend can send an HTTP POST to your endpoint whenever these events occur in real-time. We can register a webhook that listens for critical events (bounces and complaints primarily).

- **Suppressions:** Resend manages a **suppression list** internally – if an address hard-bounced or complained, future sends to it are auto-blocked (suppressed) [33] [34]. There is not a direct API to list all suppressed addresses, but the occurrence of a suppression is reflected in events (`email.suppressed` event fires if a send was blocked due to prior issues) [20].

- **Data available to read:**

- **Sent email count:** We can count the number of email records in a time range. The List Emails API supports pagination and returns recent emails first (it doesn't accept a date filter, but we can fetch and filter client-side) [35]. For instance, the dashboard might fetch the last 100 emails and count how many were sent "today" vs "yesterday" etc. Since Tabulensis likely sends moderate volume, this is fine. (If volume were huge, we'd use webhooks to accumulate counts instead.)
- **Bounces and complaints:** These are visible in two ways: (a) via email objects – any email with `last_event = "bounced"` means it bounced, similarly "complained"; or (b) via **webhook events** in real-time. The API also provides details on a bounce if you retrieve that email's record (e.g. error message, bounce type). For the dashboard, a simple counter of bounces in the last week and maybe listing the most recent bounce (with timestamp and recipient) would be useful.
- **Domain verification status:** The `domains` API returns a `status` field (e.g. "not_started", "pending", or "verified") for each domain, and the DNS records that need to be in place [19]. The dashboard can show "Sending Domain: example.com – **Verified** (DKIM/SPF OK)" or if not, highlight that with instructions. Since domain setup is usually one-time, this doesn't need frequent polling – just show current status.
- **Suppression list info:** While we cannot directly pull the entire list, we can detect if new suppressions occurred. Through webhooks, we get `email.suppressed` events when an email is blocked [20]. We can also infer suppression if an API send call returns a specific error (though in normal use, one would rely on the event). For the dashboard, it might suffice to show a count of suppressed addresses or a recent example (like "X addresses auto-suppressed after bounces; last: on Jan 5").

- **Other metrics:** If needed, Resend's API can also list **receiving emails** (if using their inbound feature) and contacts/segments if using their marketing features. But presumably Tabulensis uses Resend for transactional sending only.

- **Data not available or limitations:**

- As noted, no direct "get all suppressed emails" endpoint. One can contact Resend or check their dashboard manually for suppressions. In practice, knowing about suppressions in real-time via events is enough to take action (e.g. possibly remove an address from a mailing list – though for transactional email, usually you just avoid emailing them).

- The API returns the last event per email but not a full event history via the list endpoint. However, if we need to know if an email was delivered then later opened, we might have to retrieve that email specifically or rely on webhooks to capture intermediate events (the `last_event` will just show the latest state). For operational signals, the critical states are *bounced* and *complained*, which would be final for that email (no further events beyond suppression).
- Rate limiting: 2 requests per second per team (which is quite low but sufficient) [23] . If we naively polled the entire email list every few seconds, we'd hit this. But we don't need to – a smarter approach is to use webhooks for real-time events, and maybe poll infrequently for summary counts. Also, the list API defaults to 10 items if not paginated; we can request a larger page (maybe it allows `limit` parameter, doc suggests cursor-based pagination is available [36] [37]). We should be mindful to not page through thousands of emails on each poll. Instead, fetch recent ones.

- Plan limits: Resend has a free tier with a certain number of emails per month. API access is not restricted on free vs paid, aside from throughput. All the above endpoints are available regardless of tier (the differences are in how many emails you can send).

- **Auth model:** Resend uses API keys. We will use a **secret API key** with **Bearer** auth. The key format is `re_<random>` and it's provided in the dashboard. **Scope/Permission:** When creating a key, Resend allows two types: `full_access` (read/write everything) and `sending_access` (only send emails) [24] . To read data (list emails, domains, etc.), we must use a `full_access` key. For security, we can generate a dedicated full_access key just for the dashboard. This key should be kept locally (in an env file). In the Authorization header: `Authorization: Bearer re_xxx...` [25] . There is no granular scope beyond that. As a safety measure, if the dashboard is meant to be strictly read-only, ensure the code never calls the POST/DELETE endpoints (and one could even create a restricted key after discussing with Resend if needed, but currently "sending_access" is too restrictive and there's no "read-only" key option – so full_access it is). Logging out or rotating the key if compromised is possible from the dashboard.

- **Integration mechanism:** Ideally **use webhooks for events** and **API for aggregate/state**:

- **Webhooks:** We can set up a webhook (via `POST /webhooks` or in the Resend console) pointing to our local dashboard (if accessible, e.g. via a tunneling service or running on a server). This webhook can subscribe to relevant events: `email.delivered`, `email.bounced`, `email.complained`, `email.suppressed` as primary ones. This way, whenever an email bounces or is marked spam, the dashboard can immediately reflect it (and possibly alert the operator). Webhooks remove the need to constantly poll for these events. Resend will retry webhook deliveries for up to 24 hours on failure [26] , so it's reliable even if our dashboard is down briefly.
- **Polling:** If a persistent webhook endpoint is not feasible (say, if the dashboard truly runs only on a local machine with no incoming connectivity), then a fallback is to poll the API. One strategy: poll `GET /emails` every 5 or 10 minutes to get recent sends and check their `last_event`. Because `last_event` will change when, say, a bounce happens, you could catch events by noticing that an email's last_event went from "sent" to "bounced". However, to catch a new bounce you'd have to either fetch all recent emails or specifically fetch emails that aren't delivered yet. A simpler approach: maintain a timestamp of last poll and query only new emails since then (Resend doesn't have a direct filter by date, but we can fetch the first page which is ordered by created_at desc and stop when we reach an email we've seen).

- For aggregate info like “sent in last 24h,” we might just calculate by filtering the list, or maintain a counter that increments on each send event (Resend also sends an `email.sent` event when an email is accepted for sending [38]). Since the dashboard is internal, a slight lag in these numbers is okay.

Overall, webhooks are the high-accuracy, low-latency solution, and polling can be a periodic sanity check or fallback.

- **Coverage of desired info:** We can obtain ~95-100% of what we want:
- *Sent email count:* yes (compute from list or count events).
- *Bounces/complaints:* yes (from events or list filtering) – we will know each bounce and complaint event [39]. We might not have a built-in “% bounce rate” stat, but that’s easy to compute.
- *Domain verification:* yes (from domain status fields) – e.g. “example.com: status verified/ sending=enabled” [19].
- *Suppression list:* partial – we can’t list all suppressed addresses via API, but we know when a suppression occurs. For an operator, the main concern is knowing it happened so they don’t keep trying that address. The Resend dashboard UI would allow removal of an address from suppression if needed (the API doesn’t yet expose a remove endpoint as far as docs show). For our purposes, a count of suppressions and last suppressed address is sufficient, with perhaps a note to check Resend’s web UI for the full list if needed.

There’s no significant blind spot; the combination of API and events gives us full visibility into deliverability issues.

- **Security & privacy:** The data from Resend includes customer email addresses (in the `to` fields) and potentially email content if we fetched it (we likely will not fetch bodies for the dashboard, as it’s not needed and contains PII). We should avoid pulling content or storing any customer personal data beyond what’s needed (e.g. we might display “ bounced at 10:45AM” – that’s acceptable minimal PII, but we shouldn’t log the content of the email). The API key should be kept secret; if it’s leaked, an attacker could potentially send emails or read our sent email list. To mitigate risk, we use a dedicated key and can revoke it if something is suspicious. Additionally, if implementing webhooks, we should secure the endpoint (basic auth or a secret token in the URL) to ensure only Resend can hit it. Resend webhooks include signatures we can verify as well.

Integrating Resend is straightforward as it’s designed for developers: we just need to periodically query the needed endpoints and handle webhook POSTs. The maintenance is low – the API is stable and versionless (no versioning yet, as they note [40]). We should monitor any API announcements in case of changes.

Stripe (Payments/Billing API)

Stripe’s API is very extensive, and it will provide all the financial signals for Tabulensis’s dashboard – subscription metrics, revenue, payments issues, etc. Key aspects:

- **Available APIs:** Stripe exposes a RESTful API for all objects (customers, charges, subscriptions, invoices, payouts, etc.), and uses **webhooks** to notify of events. Specifically relevant:

- **Core REST endpoints:**
  - **Subscriptions** – `GET /v1/subscriptions` can list all subscriptions with filters (e.g. `status=active` or `status=trialing`). Each subscription object has info on its plan, status, next billing date, etc. From this we derive Active count and can sum amounts for MRR.
  - **Invoices** – `GET /v1/invoices` (or the upcoming invoice for a customer) can show if any are past due or how much is due. Failed payments are usually reflected as invoices in `payment_failed` status.
  - **Charges/PaymentIntents** – could be used to track one-time payment failures or successes, but since Tabulensis seems subscription-driven, invoices cover that. For chargebacks, Stripe has **Disputes** – `GET /v1/disputes` lists all disputes (with their status: won, lost, or open).
  - **Balance & Payouts** – `GET /v1/balance` returns the account's current balance (funds available vs pending). Pending funds are those not yet paid out (they will be in the next payout cycle) – useful for "upcoming payout" amount. `GET /v1/payouts` can list payouts, including scheduled or in-transit ones, with arrival dates.
  - There are also summary endpoints like **Stripe Sigma** (for queries) and **Reports**, but these are either paid add-ons or give CSV outputs – for an automated dashboard, we can compute what we need ourselves using the above endpoints.
- **Webhooks:** Stripe has a robust webhook system. Virtually every meaningful event in Stripe (invoice paid, invoice payment failed, subscription created or canceled, customer upgraded, payout initiated, dispute created, etc.) triggers an event that can be sent to your endpoint. By registering a webhook endpoint in Stripe's dashboard, we can subscribe to events like:
  - `invoice.payment_failed` (for failed/late payments),
  - `customer.subscription.created` or `...updated` (for new subscriptions or trials starting),
  - `customer.subscription.trial_will_end` (if we want to know when trials are about to convert or expire),
  - `charge.dispute.created` (for new disputes),
  - `payout.created` or `payout.paid` (for payout updates).
  - etc. Using webhooks means the dashboard gets notified instantly when, say, a payment fails, rather than having to poll.

- **No GraphQL or similar**: Stripe's API is purely REST (with predictable URLs and expansive filtering). But they also have a concept of **Events**: `GET /v1/events` which can list recent events (useful if you need to poll for events in lieu of webhooks). Events can be filtered by type. This is essentially an audit log of things that happened in your account over the last 30 days [41] [42].

- **Data you can read (for each desired item):**

- **ARR/MRR:** Stripe doesn't directly return "MRR," but we can calculate it. The straightforward method: query all active subscriptions and sum their monthly revenue. Each Subscription object has a `items` list with price and quantity – from that we determine how much per billing period. We need to normalize to monthly: for monthly plans, it's just plan amount; for annual plans, divide amount by 12 to contribute to MRR (or count them fully in ARR). Alternatively, one can sum the last month's invoice totals to get MRR, but since subscriptions could be mid-period, summing active subscriptions is simpler for a real-time number. **ARR** would be simply MRR × 12 (plus any annual subscriptions already counted appropriately). Because this can involve iteration over all subs, if there are many, we

might use a cached value updated by webhooks (e.g. on subscription create/cancel events, adjust the total).

- **Active subscriptions count:** Easy – use `GET /v1/subscriptions?status=active` and count the results. If few enough, one call is fine (Stripe defaults to 10 per page, but we can request up to 100 per page, or just get total_count if using the older API version; or iterate with pagination if >100). Webhook alternative: maintain a counter that increments on create and decrements on cancel. For an MVP, a single API call is fine.
- **Trial starts:** We interpret this as number of new trials (subscriptions in trial phase) started in a period. Options:
  - Use webhooks: `customer.subscription.created` events and check if `status=trialing` or if a `trial_end` is set in the future, then it's a trial start. Count those events over the period.
  - Or query `subscriptions?status=trialing&created[gte]=<start of period>` to get those created recently. The Stripe API supports filtering by creation date [43]. The webhook method is real-time and doesn't require polling multiple endpoints.
- **Failed/late payments:** Stripe signals these primarily via the **invoice.payment_failed** webhook. When an automatic payment for a subscription fails, an invoice moves to `open` or `past_due`. The dashboard should flag such cases so the operator can intervene. We can:
  - Count how many payment failures occurred in the last X days (via webhook events or by listing invoices with `status=open` or `past_due`).
  - Show a list of customers who are past due (via `GET /v1/subscriptions?status=past_due` or list invoices with `status=open`). Using webhooks, we get immediate info (and can even include the invoice ID and amount due for follow-up).
- **Disputes/chargebacks:** Stripe's dispute object is accessible. We can list all disputes with `status=open` (meaning unresolved). This gives count of ongoing chargebacks. We can also receive `charge.dispute.created` events for new ones and `...closed` when resolved. The dashboard can show count of open disputes and perhaps the latest dispute amount/reason.
- **Upcoming payouts:** Stripe typically automatically pays out daily/weekly. The **Balance** API gives two key numbers: `available` (funds that can be paid out now) and `pending` (funds waiting for payout schedule) [44] [27]. In a standard daily payout schedule, `pending` corresponds to the next payout amount. For example, if $500 is pending, that will be paid in the next cycle (often next day). We can thus label that as "Next payout: $500 on [date]". To get the date, we might use `GET /v1/payouts/upcoming` if it existed – Stripe doesn't have an explicit upcoming endpoint, but if there is a created payout with status `in_transit` or `pending`, it will show `arrival_date`. Alternatively, since payouts happen on a schedule, the date might be predictable (e.g. daily payouts arrive in 2 days by default). Simpler: display the current pending balance as the amount that will be paid out and maybe the last payout date for context (via `GET /v1/payouts?limit=1` to get the most recent payout).

Essentially, everything listed is directly queryable or derivable. Stripe's API even includes more advanced data like churn rates or lifecycle info if we wanted to calculate, but those are beyond MVP.

- **Data not available or limitations:**
- **Computed metrics:** As noted, Stripe doesn't give MRR/ARR out of the box (except via the Sigma SQL feature or reports). We will compute these. That means our numbers are as good as our calculation (for example, handling an annual subscription in MRR properly – we should take monthly equivalent).

- **Historical data:** If we wanted trends (like MRR last month vs this month), we'd have to store that or query a range of invoices. The API can list invoices by date, so we could compute, say, last month's total billed. But for MVP, point-in-time metrics are fine.
- **Rate limiting:** Stripe's rate limits are high (on the order of 100 requests/sec in live mode, 25/sec in test) [27] . Our usage will be nowhere near that. A caution: if we attempted to pull thousands of records or do a brute-force approach (like refresh all subscriptions every minute for an account with thousands of customers), we could approach limits or get a `rate_limit` error. But for a single-operator business with presumably manageable subscription counts, a few calls per minute is fine. Stripe encourages using webhooks over polling for event-driven info [44] [27] .

- **Permissions:** The API key used determines access. A standard secret key has full read/write on the account. Stripe allows creating **Restricted Keys** where you can choose read-only access to certain objects (e.g. Customers: Read, Invoices: Read, etc.) for better security. We should do this for the dashboard. One caveat: some metrics require multiple objects (e.g. to compute MRR you may need Subscriptions and Prices). We must ensure the restricted key has all necessary read permissions. This typically means enabling read access for "Orders and Products" (for prices), "Subscriptions and Customers", "Invoices & PaymentIntents", "Balance & Payouts", and "Disputes" on that key. Stripe's dashboard UI for creating restricted keys makes this straightforward, and we should double-check the dashboard data after integrating to see if something is missing (if so, adjust the key's scopes). Using a restricted key limits damage if it's ever leaked, as it can't create charges or refunds, etc.

- **Integration mechanism:**

- **Webhooks first:** We will register a webhook endpoint (Stripe calls it a webhook endpoint in the Stripe Dashboard or via API) pointing to our local dashboard (again, we might use a tunnel or run the dashboard on a server or at least use the Stripe CLI which can forward webhooks to local during development). We subscribe to relevant events. Stripe will send a JSON payload for each event. The dashboard can update in-memory counters or trigger a refresh of certain widgets on these events. Webhooks guarantee we don't miss anything and we react in real-time. For example, when `invoice.payment_failed` comes in, we increment the "failed payments" counter and maybe flag that subscription as delinquent in our local state, which the UI shows.
- **Supplement with polling:** For aggregate displays (like "current MRR" or "count of active subs"), we might do a fresh API pull on a schedule (say once a day or on dashboard load) to avoid drift. Or we can maintain a local tally updated by events:
    - E.g., start with fetching all active subs to compute MRR, then on each subscription create/cancel event adjust the MRR accordingly. This reduces API calls long-term. However, given simplicity and the relatively low cost of a single API call, an initial implementation can just pull fresh data when the dashboard is opened or when a background refresh runs (maybe daily at midnight for MRR).
- **Direct API for certain data:** Payout amount can be obtained by hitting `/balance` when the dashboard loads. That's a quick call and not something webhooked (Stripe has `payout.created` events but the balance pending is easier).

- **Polling only scenario:** If webhooks cannot be used (i.e., no exposure of local server), we could periodically poll `/events` with a filter (e.g. all events of type `invoice.payment_failed` since last check). But this is essentially re-creating webhook functionality with more complexity and delay. It's possible (Stripe allows filtering events by type and creation time [45]), but not ideal. For an internal tool, setting up webhooks via something like an AWS Lambda or a small cloud endpoint that

then relays to the local app (or writes to a file the local app reads) is an option if we wanted to avoid opening the local network. However, assuming the operator can run something like `stripe listen` (Stripe CLI) and forward to the local app during use, we can get webhooks working during active periods. We will prioritize webhooks in design for timely updates.

- **Feasibility and completeness:** Stripe's API will provide **100%** of the desired information:

- Every data point listed (MRR, counts, events) is accessible as described.
- The only "effort" is combining data to create the metric (which we can handle in code).
- There are no known gaps – even things like "trials started" which might not be readily visible in the Stripe Dashboard can be obtained via events or filters.
- Because Stripe handles sensitive financial data, we should be extremely careful with security. Ensure the secret key (or restricted key) is not exposed (no client-side usage, only server-side). Also verify webhook signatures (Stripe provides a signing secret and libraries to check that the payload is truly from Stripe). This prevents any malicious post to our endpoint from spoofing a Stripe event.
- Privacy: Stripe data includes customer identifiers, maybe emails depending on how subscriptions are set up, and amounts. This is all for internal use, but the dashboard should not expose full customer PII more than necessary. For example, showing "3 failed payments (Customer: Alice, Bob, ...)" might be okay internally, but maybe just the fact that there are fails is enough. At least do not log or expose raw API responses containing full customer records.

In summary, Stripe integration is very powerful. We will leverage webhooks for instantaneous updates on the most critical events (failed payments, new signups, etc.) and use direct API calls for periodic or on-load aggregation (like computing total MRR or listing current open disputes). Stripe's library support is excellent as well (we can use official client libraries which handle a lot of boilerplate like pagination or event signature verification, reducing maintenance burden). Because this is read-only and our tool is not customer-facing, we avoid any write operations (like no accidental charges). The dashboard essentially mirrors Stripe's own Dashboard data but in a custom unified view.

Proposed MVP Dashboard

Bringing it all together, we propose an MVP dashboard that is organized into sections or "widgets" for each category of interest. Below are the specific widgets and the data sources/API calls to power them:

- **Fastmail – Inbox Status:**
- **Unread Count:** Display the number of unread emails in the support inbox. *Data source:* JMAP `Mailbox/get` for Inbox – this returns `unreadEmails` count [1]. For example, call `POST https://api.fastmail.com/jmap/` with body to get the mailbox state. The response includes something like `"totalEmails": X, "unreadEmails": Y` for Inbox. This Y is the unread count.
- **Pending Replies by Age (SLA buckets):** Show how many unread (or unreplied) emails are older than 1 day, 3 days, 7 days, etc. This helps the operator see overdue emails. *Data source:* JMAP `Email/query` with conditions or filtering client-side. We can query all `isUnread=true` in Inbox, then for each get the `receivedAt` timestamp. By comparing to now, categorize counts. (JMAP can sort by date, or we can fetch all unread message ids and then fetch headers for the oldest few to list specifically.)
- **Oldest Unread Email Subjects:** Optionally list the subject lines of the 1-3 oldest unread emails waiting. This provides context on what's pending. *Data source:* JMAP `Email/get` for those specific

email IDs to retrieve `subject` (and maybe `from` snippet). This requires an additional call or an expansion of the query call (JMAP allows a `collapseThreads` or getting a preview, etc.). We will only fetch minimal fields to avoid large data. No full bodies.

Implementation: A single JMAP request can retrieve mailbox counts and maybe do a query for unread IDs (using `query` and `get` in one batch). The results update this widget. Poll this every 5 minutes or on demand. This is read-only and low-risk.

- **Cloudflare – Website & Security:**
- **Traffic (Last 24h):** A summary of website traffic, e.g., "XYZ requests in last 24h, ABC unique visitors, DEF MB bandwidth, GHI cache hit ratio". *Data source:* Cloudflare GraphQL Analytics [6]. We will craft a GraphQL query for our zone ID to get: sum of requests, unique count, sum of bytes, and perhaps cache hit percentage, over the past day. Alternatively, use the Zone Analytics REST (if still operational) for simpler data [29], but GraphQL is preferred since the REST may be deprecated [46]. The GraphQL query might look like:

```
{ viewer { zones(filter: {zoneTag: "<ZONE_ID>"}) {
     httpRequests1dGroups(limit: 1, filter: {datetime_gt: "<24h ago>"}
aggregate: { count, uniqueVisitors, bytes, cachedBytes } } } }
```

  – which returns aggregated counts. The dashboard can show these numbers and maybe a small sparkline or trend if we query time-series (GraphQL could give hourly data points for a chart).
- **Worker Errors:** "Worker Errors in last 24h: X" (or last hour). If Tabulensis uses Cloudflare Workers (for a serverless function or to serve the app), this metric shows if code is failing. *Data source:* Cloudflare GraphQL **Workers** analytics [8]. Query the worker's metrics grouping by `status` (success vs error) for the timeframe. For example:

```
{ viewer { accounts(filter: {accountTag: "<ACCOUNT_ID>"}) {
     workersInvocationsAdaptive(limit: 10, filter: {scriptName: "<NAME>",
datetime_gt: "-24h"}) {
          sum { errors, requests } } } } }
```

  – This yields total requests and errors. We display the count or percentage of errors. If no Workers in use, this widget can be omitted or repurposed for another Cloudflare stat (like "origin errors" – e.g., 5xx responses).
- **Deployment Status:** Indicate the status of the last site deployment. For example, "**Site Deployment:** Success (Jan 12, 14:05 UTC)" or "Failed". *Data source:* **Cloudflare Pages API** – `GET /accounts/<ID>/pages/projects/<project>/deployments?per_page=1` returns the latest deployment record, including `latest_stage.status` [15] and timestamps. We check that status ("success" or "failure"). If using Workers instead of Pages, we could instead show a simpler status like "Last published at <time>" (the publish time would come from our own deployment script logs, since CF doesn't expose an explicit "deployment" for workers aside from listing script versions via API). For MVP, assume Pages is used for any static site.
- **DNS/Config Health:** This could show a green check if everything is okay (domain is active on Cloudflare, DNSSEC enabled if required, etc.). *Data source:* Cloudflare REST – `GET /zones/<ZONE_ID>` gives overall status of the zone (should be "active") and whether DNSSEC is on, etc. Also,

if we expect certain DNS records (like MX for email, CNAME for a subdomain), we could retrieve `GET /zones/<ZONE_ID>/dns_records?name=...` to ensure it exists. For now, perhaps just confirm "Domain is active on Cloudflare" from the zone status. If Cloudflare's status API indicates any incidents for our zone (not likely accessible per zone, they have a global status API for outages), that could be integrated, but that's advanced.

- **Security Events:** "Firewall events in last 24h: N blocked or challenged". *Data source:* Cloudflare GraphQL – query `firewallEventsAdaptiveGroups` for count of events over last day [7]. For example:

```
{ viewer { zones(filter: {zoneTag: "<ZONE_ID>"}) {
     firewallEventsAdaptive(limit: 1, filter: {datetime_gt: "-24h"}) {
         count
     } } } }
```

This yields how many firewall events occurred. We might split by action (e.g. how many blocked vs challenged vs JS challenge) if desired, but initially just the total "threats mitigated" is fine. The widget can show that number; if >0, maybe allow clicking to see the most recent event details (we could fetch the last 1-5 events via GraphQL or via the (undocumented) firewall events REST if it exists).

Update frequency: Traffic and security can be polled every 5 or 10 minutes. Deployment status could be checked more frequently only around deployment times – but since the operator likely knows when they trigger a deployment, a manual refresh button could suffice. DNS health is static, check daily.

- **Resend – Email Delivery:**
- **Emails Sent (Today):** Display the count of emails sent today (or this week) via Resend. *Data source:* Resend `GET /emails` with filtering client-side. For instance, fetch the first 50 sent emails sorted by created_at; count how many have `created_at` timestamp after midnight. Alternatively, maintain a running daily count via events (`email.sent` events increment a counter). This gives a quick view of volume (to ensure, for example, that scheduled emails are indeed going out).
- **Bounces & Complaints:** Show the count of bounced emails and spam complaints in the last day/week, and possibly list the most recent one. E.g., "**Bounces:** 1 (john@foo.com on Jan 10) – **Complaints:** 0". *Data source:* Resend webhooks. Each `email.bounced` event can increment a bounce counter and store the email/address. Similarly for `email.complained`. We could also query the email list for any with `last_event="bounced"` in recent history as a backup. The dashboard should highlight if there's a spike or any bounce at all (since even one bounce might warrant attention – perhaps the email address was bad). Because Resend auto-suppresses hard bounces, a bounce essentially means that address is now on suppression.
- **Domain Status:** For each sending domain (likely just one domain), show verification status. E.g., "**Sending Domain:** example.com – Verified ✅" or if not verified, show "Not verified ⚠ (DNS records incomplete)". *Data source:* `GET /domains` (which returns an array of domains with their status). If not verified, we can even parse the `records` field which shows which DNS records are missing (`status: "not_started"` or `pending`) [47] and display a hint (e.g., "DKIM record not found"). This is mostly static after initial setup, but it's good to have on the dashboard until it's verified.
- **Suppression List Alert:** Perhaps a small indicator like "Suppressed addresses: X" where X is the number of addresses currently suppressed due to bounces/complaints. We might not know X directly via API, but we can track how many unique addresses have been suppressed (Resend's docs suggest addresses get unsuppressed only if manually removed). Alternatively, simply flag if any new

suppression occurred recently: e.g., “Suppression triggered: Yes – 1 address newly suppressed this week”. *Data source:* `email.suppressed` webhook events. Each such event likely contains the email that was suppressed. We log it and can present the count. The operator could then login to the Resend dashboard to remove if needed or just be aware.

- Optionally, **Delivery Rate:** We can compute a simple delivery success rate = (sent - bounced - suppressed) / sent, for recent emails. If this drops, it signals an issue (like many bounces). This could be a small percentage display.

Update mechanism: Webhooks will update bounces/complaints in real-time. The sent count can also be incremented on `email.sent` events (these fire when an email is accepted for sending). Domain status is static; check it on startup or once a day. The suppression count updates when suppression events come in.

- **Stripe – Revenue & Billing:**
- **MRR (Monthly Recurring Revenue):** Display the current MRR (in USD or relevant currency). E.g., “**MRR:** $2,500”. *Data source:* Calculate from active subscriptions. We can either:
  - Use the Stripe API: `GET /v1/subscriptions?status=active` and sum `subscription.items[].price.unit_amount * quantity` for those that bill monthly. For annual subscriptions, divide unit_amount by 12 (assuming unit_amount is per billing period). If there are different intervals, we normalize all to monthly. Another approach is to sum the `monthly_payment` field if using Stripe’s Scheduled Query (not using Sigma here). We’ll implement the calculation ourselves.
  - Or use cached data updated by events: whenever a subscription is created, upgraded, downgraded, or canceled (`customer.subscription.updated` and `...deleted` events), recalc or adjust MRR. For MVP, a direct pull once a day is fine. This metric doesn’t need to update multiple times per day unless new signups are very frequent (the webhooks for subscription changes can trigger an instant update anyway).
- **ARR:** Could be shown as 12×MRR if needed (“ARR: $30,000”), or omitted in favor of just MRR which is more immediately useful for a monthly cadence business. We include it since requested. Once MRR is computed, ARR is trivial math on it.
- **Active Subscriptions:** “**Active Subscribers:** N”. *Data:* `GET /v1/subscriptions?status=active` and count, or maintain a counter updated by events (subscription created +1, canceled -1). This is straightforward. Possibly also show “(+X trials)” if there are trialing subs too, but that’s next item.
- **Trials Started (This Week):** Show how many new trial subscriptions began in the defined period (week or month). *Data:* We can use a webhook `customer.subscription.created` which will tell us if a subscription starts in trial (the object will have `status=trialing` and a `trial_end`). We increment a weekly counter. Or query `GET /v1/subscriptions?status=trialing&created[gte]=<week_start>` each time – either works. The dashboard might say, for example, “Trials started this week: 3” which tells the operator if top-of-funnel is active.
- **Payment Failures:** “**Failed Payments:** X in last 7 days”. Also possibly list the most recent failed invoice (customer or amount) so the operator can follow up. *Data:* Webhook `invoice.payment_failed` events – we count them and capture details. Or query `GET /v1/invoices?status=open` to see how many are open (meaning attempts failed and they’re in dunning). The webhooks approach is better for a rolling count. The dashboard can display the count and maybe highlight if any are currently past due (which likely there are if a failure happened and hasn’t been resolved). We might even list customer names or emails for those failures for convenience (Stripe event data includes customer ID; we could fetch the customer email if needed, but to minimize PII maybe just indicate “2 customers have failed payments”).

- **Disputes/Chargebacks:** "**Open Disputes:** Y ($Z total)" where Y is count of ongoing chargebacks. *Data:* `GET /v1/disputes?status=open` will list them. Or listen to `charge.dispute.created` (increment) and `...closed` (decrement) events. The dashboard might show the count and sum of disputed amounts (dispute objects contain the amount). This alerts the operator to potentially reach out or check those cases in Stripe.
- **Upcoming Payout:** "**Next Payout:** $X on [date]". *Data:* `GET /v1/balance` provides `pending` amounts by currency. For a single currency account, pending[0].amount = X (cents). Stripe doesn't provide the exact date via this endpoint, but typically payouts in pending will be paid on the next schedule (if daily, usually tomorrow). For more precision, we could fetch the most recent payout's arrival date and assume the next is one payout cycle from then, but this gets complicated for different schedules. The dashboard can approximate or just show "Pending balance: $X" which implicitly will be in next payout. Alternatively, `GET /v1/payouts?limit=1&status=in_transit` might return a payout that's initiated and arriving in e.g. 2 days with an arrival_date. We can experiment: if an upcoming payout is already created, we can show its date. If on manual payouts, then pending won't auto-payout; but Tabulensis likely uses auto.

Additionally, we might show **ARR** as mentioned, or a monthly revenue run-rate (which is basically MRR). We could also show "Month-to-date revenue" by summing all successful payments since the 1st of the month via `GET /v1/charges` or invoices – but that wasn't explicitly asked and might be overkill. If needed, it's doable with one API call filter by date.

Update mechanism: - Use **webhooks** for event-driven parts: e.g. when a subscription is created or canceled, update active count and MRR. When an invoice fails, increment the failure count and perhaps add to a "needs attention" list. When a dispute comes in, add it. - For totals like MRR that might be easier to recalc, we can recalc daily or on dashboard load by pulling subscriptions fresh. Because MRR/active subs are just snapshots, recalculating them once a day (or in response to any sub change event) is fine. - Payout info could be pulled daily or every few hours (since it's not urgent). - The dashboard section can combine these: e.g. a "Revenue Summary" panel that shows MRR, active subs, trials, and a "Billing Alerts" panel for fails and disputes.

All of these use read-only operations. A note: summing money amounts from Stripe requires correct currency handling (if multi-currency, MRR should ideally be in one currency – likely Tabulensis charges in one currency). Assuming USD only for simplicity. Stripe amounts are in cents, so we'll divide by 100.

Each widget above will be mapped to specific API calls as described, and we'll implement them ensuring we handle pagination (if needed) and errors (with safe fallbacks, e.g. if a GraphQL query fails, perhaps try a simpler one or skip that update).

Recommended Next Steps

To move forward, here is an ordered list of next steps, focusing on delivering value quickly and securing the integrations. The first steps are doable within <2 hours:

1. **Get API Credentials & Setup Environment:** Immediately generate the required API credentials:
2. Create a Fastmail API token (with JMAP mail access) from your account settings [2].
3. Create a Cloudflare API Token: go to **My Profile > API Tokens**, use the "Read Analytics" template and add DNS if needed [10]. Restrict to your zone.

4. Create a Resend API key (full_access) in the Resend dashboard [24].

5. Create a Stripe restricted API key in Stripe Dashboard: give read permissions to all relevant objects (Customers, Subscriptions, Invoices, Balance, etc.). Store all these secrets in a local `.env` file or config (not in code repo). Set up your development environment to load these.

6. **Prototype Data Fetch (CLI or Script):** Write a simple script (using curl or a library in your language of choice) to test each integration:

7. Fetch Fastmail Inbox unread count via JMAP (you can use their example from docs or a small library). Confirm it returns expected values.
8. Query Cloudflare GraphQL for a simple metric (e.g. past 24h request count) [48]. If GraphQL is complex to construct, start with a simpler REST call like zone analytics (it might return data for last 24h by default).
9. List the last 5 sent emails from Resend via `GET /emails`. Check parsing of JSON, and note the `last_event` field [16].
10. Fetch Stripe data for one item, e.g. call `GET /balance` to see pending/available, or `GET /subscriptions?limit=1` to ensure the key works. This step ensures all keys and endpoints work. It should take under 2 hours using tools like Postman or simple Python scripts. Save sample responses for reference (but avoid logging sensitive data).

11. **Set Up Webhook Endpoints (locally):** Decide how to handle webhooks locally. E.g., use Flask/FastAPI or Node Express to create endpoints `/webhook/resend` and `/webhook/stripe`. Implement logic to handle a basic event (just log receipt for now).

12. Use a tunneling service (ngrok or Cloudflare Tunnel) to expose your local webhook endpoints to the internet temporarily for testing.
13. Configure Resend Webhook: `POST /webhooks` with events ["email.bounced", "email.complained", "email.suppressed", "email.delivered"] pointing to your endpoint (or set it up in their dashboard UI).
14. Configure Stripe Webhook: In Stripe Dashboard, add an endpoint for relevant events (you can select a few or use wildcard and filter in code). Stripe will provide a signing secret – note it.
15. Send test events: Resend allows sending a test email or you can trigger a bounce by emailing an invalid address. Stripe you can trigger test mode events (Stripe CLI can send a `stripe trigger invoice_payment_failed` event in test mode, for example).

16. Verify your local server receives and logs these events. Implement signature verification for Stripe to ensure security (Stripe provides code for this). This step might take a couple of hours to fully test, but it's crucial for real-time data.

17. **Choose Dashboard Architecture & Scaffold:** Given the successful tests, decide on the implementation:

18. If you choose a quick TUI/console approach: scaffold a Python script that on run, fetches all APIs and prints a summary. You could use a library like `rich` to format nicely and even update dynamically. This is quickest to implement.

19. If you prefer a web UI: scaffold a minimal Flask app with routes (one for main dashboard page, one for each webhook). Use a simple HTML template or even plain text initially to display data. In either case, structure the code into modules by service for cleanliness. Plan to run this app on your machine (or a server if needed later).

20. **Implement Service Integrations (Iterative):** Start coding each integration one by one, verifying as you go:

21. Fastmail: implement a function to fetch unread count and maybe oldest unread. Test it and format the output.
22. Cloudflare: implement a function to fetch GraphQL analytics (perhaps start with a simple query for request count). If GraphQL is too time-consuming to figure out, use the zone analytics REST as a placeholder, then improve it. Also implement a function for Pages deploy status if relevant (you can test by deploying a trivial change to see API response).
23. Resend: implement fetching recent emails and parsing counts. Also set up the webhook handler to update a global state (like increment bounce count when a bounce event comes in). Use threading or async if your framework requires (Flask can handle webhook POST while main thread is doing something else, but we might use a simple global state dict for counts).

24. Stripe: implement pulling subscription count and MRR. This might involve retrieving subscriptions list – be mindful of pagination if >100 (unlikely initially). Implement webhook handling: on `invoice.payment_failed`, mark that invoice/customer in a dict of failures; on `customer.subscription.created`, increment active count (and add to MRR calculation). Test each integration piece independently (e.g. call the function and print results, simulate webhook calls by posting stored JSON to your endpoint).

25. **Aggregate and Display:** Once the pieces are fetching data, integrate them into the dashboard display. Arrange the output as per the categories:

26. Perhaps use a simple HTML page with sections for Fastmail, Cloudflare, etc., populating values from the latest fetched data. Or if TUI, print sections with headings and values.
27. Make sure to format numbers nicely (e.g. currency with `$` and two decimals for Stripe amounts).
28. Implement simple color-coding or icons for status (e.g. green check if no issues, red X if something failed). For instance, if Cloudflare firewall events > 100 in a day, maybe highlight that section.

29. Keep the UI minimal but clear (the user specifically values readability and scanning).

30. **Schedule/Polling Mechanism:** Set up background tasks or a loop to refresh data at appropriate intervals:

31. You can use Python's `sched` or simply a while loop with `time.sleep()` in a thread to periodically update each service's data. For a web app, you might not need real polling if you refresh page manually, but auto-refresh can be nice (maybe use meta refresh tag or small JS for periodic refresh if web).
32. Ensure that webhook events update the in-memory data immediately. You might need to lock around data structures if multi-threaded.

33. For example, you might poll Fastmail and Cloudflare every 5 minutes, Resend maybe every 10 (with webhooks handling immediate events), Stripe perhaps every 15 (most data comes via webhooks anyway).

34. In a cron-script scenario, you'd schedule separate cron jobs for each service or one job that aggregates all – but since we want a unified view, running a persistent process is easier.

35. **Security Pass:** Before deploying it even internally, do a quick audit:

36. Remove or secure any sensitive logging (don't log full API responses or secrets).
37. In the code, ensure API keys are not exposed (no serving them to client). In a web app, all calls should be server-side.
38. For webhooks, verify payloads (we did for Stripe; Resend's webhook can be authenticated via a shared secret in the URL since they don't sign events as far as docs show).

39. Consider storing minimal persistent data. If you log events to a file (for debugging), be cautious it might contain emails or customer IDs – secure or disable those logs for production use.

40. **User Acceptance & Iteration:** Run the dashboard for a few days to see if it surfaces the intended "high-ROI signals." Solicit feedback from the operator (yourself, in this case):

41. Are the time intervals and thresholds appropriate? (Maybe you want last 7 days traffic, not 24h, etc. Adjust GraphQL queries accordingly.)
42. Is any important signal missing? (Perhaps you realize you want to see "new signups today" – which you can get from Stripe or your app DB, but wasn't initially listed. You could add that if needed.)
43. Fine-tune visuals for clarity: e.g. highlight "0" bounces in green (all good) vs a nonzero in red.
44. If any data seems off, double-check API logic (especially MRR calc or if any events were missed).

45. Document how to run the dashboard (so it's not just working on your dev machine – ensure it can start on your machine's startup or via a simple command).

46. **Future Enhancements Planning:** Note down ideas for improvements that didn't make MVP. For example:

    - Storing history to show trends (maybe the dashboard could show a small trend chart of traffic or MRR over time – would require saving data daily).
    - Adding other integrations (if Tabulensis uses other services).
    - Deeper Cloudflare analytics like performance (TTFB or Core Web Vitals if available via API).
    - Perhaps a mobile-friendly view or text-alert if certain thresholds exceeded (e.g. send SMS via Twilio if a payment fails – beyond dashboard but related). With MVP done, these can be prioritized in the backlog.

By following these steps, within a day or two we should have a functional internal dashboard. The initial tasks (getting keys and testing calls) are very quick and de-risk the project. Using official docs (cited above) ensures we implement against the correct endpoints. We should verify any unclear aspects (e.g. GraphQL query syntax or Stripe restricted key permissions) by referencing docs or doing small experiments (for instance, run a GraphQL query in Cloudflare's GraphiQL IDE with your token to make sure the scope is sufficient, adjusting the token if needed).

Finally, ensure the system is **local-only** as intended: no data should be sent or stored externally beyond pulling from the vendor APIs. If using any cloud for webhooks (for example, an AWS Lambda to relay), be mindful of data passing through and secure it (or avoid it by running the dashboard on a machine that can receive webhooks directly).

Appendix: Links (Official Documentation)

- **Fastmail API Documentation (Developers):** Fastmail's guide to using JMAP and other protocols, including authentication and scopes [1] [2].
- **JMAP Spec – RFC 8621 (Mail capabilities):** The official JMAP mail specification [5] (useful for understanding methods like Email/query).
- **Cloudflare GraphQL Analytics API Docs:** Official documentation for querying analytics with GraphQL [6] [7], including examples for firewall events and metrics queries.
- **Cloudflare API Rate Limits:** Cloudflare Fundamentals doc on API limits [14] [9] (1,200 req/5min global, GraphQL cost limits).
- **Cloudflare Pages REST API:** Guide on using the Pages API for deployments [15] [30] – covers required token scopes and sample requests for deployments.
- **Resend API Reference – Introduction:** Overview of Resend's REST API (base URL, auth, rate limits) [25] [23].
- **Resend API Reference – Emails:** "List Sent Emails" endpoint with response example [16] and email object fields (`last_event` etc.).
- **Resend Webhook Events:** List of Resend event types (delivered, bounced, complained, suppressed, etc.) [17] [18] for configuring and handling webhooks.
- **Stripe API Reference:** General reference for Stripe's API (customers, subscriptions, invoices, etc.) – e.g., Stripe API – Events (listing events) [41] [42].
- **Stripe Developer Docs – Using Webhooks:** Guide emphasizing using webhooks over polling and note on rate limiting [44] [27].
- **Stripe Support – Rate Limit Policy:** Stripe support article confirming 100 req/sec (live) and 25 req/sec (test) limits (for awareness on polling) [44] [27].
- **Stripe Best Practices – Restricted Keys:** Stripe documentation on creating restricted API keys and managing secret keys (to implement least privilege) [49] [50].

(All links above reference official documentation pages for accuracy and further details. It's recommended to review the docs for any updates or changes in API behavior. Plan-tier nuances, such as data retention or feature availability, are noted where applicable, but should be verified against your specific account if any doubt.)

---

[1] [2] [3] [4] [5] [28] API Documentation | Fastmail
https://www.fastmail.com/dev/

[6] [13] GraphQL Analytics API · Cloudflare Analytics docs
https://developers.cloudflare.com/analytics/graphql-api/

[7] [12] Querying Firewall Events with GraphQL · Cloudflare Analytics docs
https://developers.cloudflare.com/analytics/graphql-api/tutorials/querying-firewall-events/

[8] Querying Workers Metrics with GraphQL · Cloudflare Analytics docs
https://developers.cloudflare.com/analytics/graphql-api/tutorials/querying-workers-metrics/

9 14 Rate limits · Cloudflare Fundamentals docs
https://developers.cloudflare.com/fundamentals/api/reference/limits/

10 11 30 REST API · Cloudflare Pages docs
https://developers.cloudflare.com/pages/configuration/api/

15 Cloudflare API | Pages › Projects › Deployments › list
https://developers.cloudflare.com/api/python/resources/pages/subresources/projects/subresources/deployments/methods/list/

16 35 List Sent Emails - Resend
https://resend.com/docs/api-reference/emails/list-emails

17 18 20 38 39 Event Types - Resend
https://resend.com/docs/webhooks/event-types

19 47 Retrieve Domain - Resend
https://resend.com/docs/api-reference/domains/get-domain

21 22 23 25 36 37 40 Introduction - Resend
https://resend.com/docs/api-reference/introduction

24 Create API key - Resend
https://resend.com/docs/api-reference/api-keys/create-api-key

26 Webhooks Ingester - Resend
https://resend.com/changelog/webhooks-ingester

27 44 docs.stripe.com
https://docs.stripe.com/payments/payment-methods

29 Zone Analytics to GraphQL Analytics - Cloudflare Docs
https://developers.cloudflare.com/analytics/graphql-api/migration-guides/zone-analytics/

31 Configure an Analytics API token · Cloudflare Analytics docs
https://developers.cloudflare.com/analytics/graphql-api/getting-started/authentication/api-token-auth/

32 48 CF API: no more analytics for Free plans? : r/CloudFlare
https://www.reddit.com/r/CloudFlare/comments/1l97v6l/cf_api_no_more_analytics_for_free_plans/

33 34 Email Suppressions - Resend
https://resend.com/docs/dashboard/emails/email-suppressions

41 42 43 45 docs.stripe.com
https://docs.stripe.com/api/events/list

46 Zone Analytics Colos Endpoint to GraphQL Analytics - Cloudflare Docs
https://developers.cloudflare.com/analytics/graphql-api/migration-guides/zone-analytics-colos/

49 Best practices for managing secret API keys - Stripe Documentation
https://docs.stripe.com/keys-best-practices

50 Restricted API key authentication - Stripe Documentation
https://docs.stripe.com/stripe-apps/api-authentication/rak