ChatGPT

Scalable Ingestion Architecture for Zeek

Goal: Design a cost-efficient, TypeScript-first pipeline to reliably ingest **10,000+ sources per day** (YouTube transcripts, podcasts, articles, RSS feeds, social media, etc.), with a clear separation between the ingestion **engine** and the **API**. The solution should avoid Python and long-running jobs on Vercel, favoring alternatives that are easier to maintain and observe (with good logging/monitoring). We also highlight open-source services/APIs that can broaden ingestion coverage. Below is a practical architecture guide with diagrams, example tools, and implementation tips.

High-Level Architecture Overview

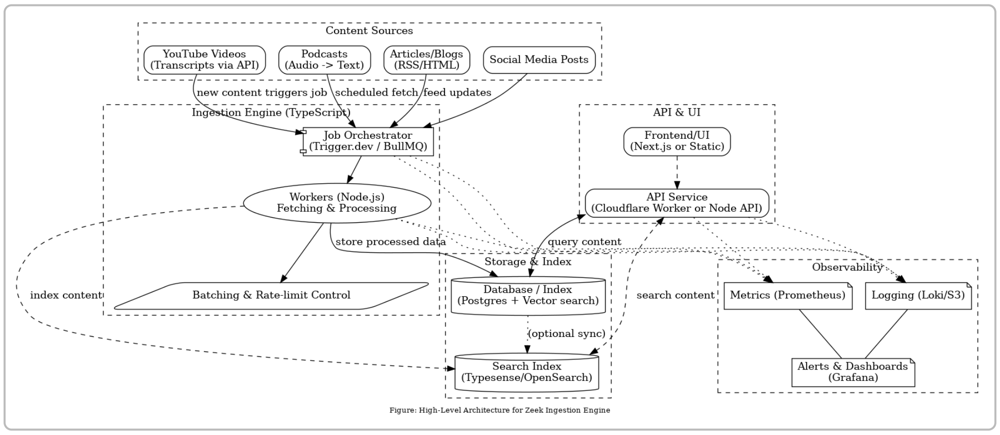
Content Sources
YouTube Videos (Transcripts via API)
Podcasts (Audio -> Text)
Articles/Blogs (RSS/HTML)
Social Media Posts
new content triggers job
scheduled fetch
feed updates
Ingestion Engine (TypeScript)
Job Orchestrator (Trigger.dev / BullMQ)
Workers (Node.js) Fetching & Processing
store processed data
Batching & Rate-limit Control
index content
API & UI
Frontend/UI (Next.js or Static)
API Service (Cloudflare Worker or Node API)
query content
search content
Storage & Index
Database / Index (Postgres + Vector search)
(optional sync)
Search Index (Typesense/OpenSearch)
Observability
Metrics (Prometheus)
Logging (Loki/S3)
Alerts & Dashboards (Grafana)
Figure: High-Level Architecture for Zeek Ingestion Engine

Figure: High-Level Architecture for Zeek Ingestion Engine. Content sources feed into a TypeScript-based Ingestion Engine that runs background jobs to fetch/parse content. Processed data is stored in a database (and optionally a search index). A separate API service (e.g. Cloudflare Worker or Node server) queries this data to serve client requests. Observability components (logging, metrics) collect data from both engine and API for reliability.

Key Components

- **Content Sources:** Thousands of external sources (RSS feeds, YouTube channels, podcasts, etc.) provide new content to ingest each day.
- **Ingestion Engine (TypeScript):** A backend service responsible for scheduling and processing fetch tasks. It includes a **job orchestrator** (task queue and scheduler) and multiple **worker** processes to handle fetching, parsing, and processing content concurrently.
- **Storage & Index:** A database (e.g. PostgreSQL) to store raw and processed data (transcripts, text, metadata). Optionally, a specialized search index (like Typesense or OpenSearch) can be used for full-text search or vector similarity queries on ingested content.
- **API & UI Layer:** A separate web service that exposes APIs (GraphQL/REST) for clients. This could be a Cloudflare Workers script or a Node/Next.js service running on a provider like Fly.io. It reads from the database/index to serve queries (e.g. search or retrieval of ingested content) without performing heavy processing itself.

- **Observability:** Logging, monitoring, and alerting infrastructure to ensure the system runs reliably. All engine and API components emit logs and metrics to a monitoring stack (Grafana, Prometheus, Loki, etc.), enabling performance tracking and quick error detection.

This separation ensures the **ingestion engine** can perform heavy lifting and recover from failures independently, while the **API** remains fast and responsive for end-users. Next, we dive into each part and suggest technologies and strategies.

Ingestion Engine Design (TypeScript-First)

The core of the system is a **TypeScript-based ingestion engine** that can handle thousands of jobs per day with high reliability. This engine will continuously gather content from various sources, and it must be scalable and resilient (e.g. handle retries, rate limits, batching). Key design points:

- **Node.js Workers and Job Queue:** Use Node/TypeScript for writing the workers that fetch and process data. Node's asynchronous model is well-suited for I/O-heavy tasks like downloading feeds or calling APIs. A job queue system is critical to coordinate work and prevent overload. Instead of `pg-boss` (a Postgres-based queue library), consider more robust and visible alternatives:
- **Trigger.dev** – A modern open-source TypeScript job orchestration library with built-in UI and scheduling. It integrates with Node apps to define background jobs with retries, delays, and even cron schedules [1] [2]. Midday (an open-source SaaS) uses Trigger.dev to handle its background tasks at scale, finding it a "game changer" for reliability [3]. Trigger.dev can be self-hosted (it uses a Postgres database under the hood, and can run on Supabase) and provides real-time monitoring of tasks (including status, logs, and Slack alert integrations) [2]. This greatly improves visibility and debugging compared to pg-boss.
- **BullMQ (Redis Queue)** – A popular Node job queue backed by Redis. It's high-performance (100k+ jobs/sec) and supports features like delayed jobs, concurrency control, rate limiting, and retries [4] [5]. BullMQ has a simple API for defining workers and producers in TypeScript, and you can pair it with a UI dashboard (e.g. **Bull Board** or **Arena**) to inspect queues and jobs in real time. This is easier to set up than pg-boss and scales horizontally by adding Redis and worker nodes [6]. Each worker process will pull jobs from Redis, so multiple workers can share the load.
- **Temporal (TypeScript SDK)** – A powerful open-source workflow engine. Temporal offers durable workflows with a TypeScript SDK, ensuring every step completes even across restarts. It has a built-in web UI for visibility and supports complex orchestrations and human-in-the-loop steps [7] [8]. Temporal could be overkill for initial needs (it requires running a Temporal server or using their cloud), but it's an option if you need very advanced orchestration or built-in tracing. For <10k jobs/day, simpler solutions like Trigger.dev or BullMQ are usually sufficient and much easier to deploy.
- **Cloudflare Workers + Queues** – If you prefer a serverless approach, Cloudflare has beta support for **Queues** and Cron Triggers. You could enqueue URLs or tasks into Cloudflare Queues and have Workers process them. This is highly scalable and removes server management. However, each Worker invocation has CPU time limits (50ms CPU time by default, up to 30s for scheduled or Durable Object jobs) which might be challenging for heavy processing. A hybrid approach is possible: use Cloudflare Workers to **dispatch** tasks (e.g. read an RSS feed and enqueue items) and a Node service to perform heavier processing. Given the budget and complexity, a small Node service with a queue may be simpler to reason about and debug.

Recommendation: Use **Trigger.dev** for a seamless TypeScript workflow and great visibility, or **BullMQ + Redis** if you prefer a proven queue system. Both options are easier to set up than pg-boss and include monitoring capabilities out-of-the-box (Trigger.dev's dashboard or BullMQ's UI) for tracking job status.

- **Scheduling & Batching:** Schedule source fetches in a distributed manner to avoid spikes. For example, use a cron scheduler (via Trigger.dev or BullMQ's repeatable jobs) to spread out feed updates over time. Midday's engine schedules its bank data sync jobs at **randomized times within each day to avoid API rate limits** [9] – Zeek can adopt a similar approach for daily source updates. If 10,000 sources must be checked per day, consider dividing them into hourly or minute-level batches (e.g. ~420 sources/hour, ~7 per minute) to smooth load. The job orchestrator can enforce a **concurrency limit** (e.g. only X fetches running in parallel) and a **rate-limit policy** for external APIs [9] [2]. Many queue systems support these features (Trigger.dev tasks can have concurrency settings and backoff retries; BullMQ supports per-queue or per-job-type rate limits and global concurrency settings [10]).
- **Batching strategy:** If certain tasks are lightweight individually, you can batch them to reduce overhead. For example, instead of launching 10000 separate HTTP requests at once, a worker could fetch feeds in batches of 50–100 in one job cycle (iterating through them). Alternatively, group sources by type: e.g. one job fetches *all* RSS feeds for a category or one job pulls updates for 20 YouTube channels via a single API call if possible. **Batch processing** is one reason Midday chose Trigger.dev – it easily handles embedding generation in batches for their receipts processing [11]. Aim to maximize throughput while respecting external API limits. In practice, this means using asynchronous fetching (e.g. `Promise.all` in Node for parallel network calls) and possibly slight delays between batches to avoid bursts.

- **Retry & Failover:** The engine should automatically retry transient failures (with exponential backoff). Use the queue's retry features to requeue failed jobs a few times before giving up. For example, BullMQ allows setting `attempts: 3` with backoff so a job will retry with delays if it fails [12]. Also consider a **Dead Letter Queue** pattern – failed tasks after max retries can be moved to a separate queue for manual review or later reprocessing so you don't lose data.

- **Content Processing:** Each source type might require a different processing step:

- *YouTube videos:* Use YouTube's Data API to fetch video metadata and transcripts if available (YouTube provides captions via API if the video has subtitles/CC). Alternatively, for audio-only transcripts you might integrate a speech-to-text service (but that can be expensive for many hours of content). If transcripts aren't directly accessible, consider tools like `youtube-dl` / `yt-dlp` to retrieve captions, or third-party APIs. These tasks could be offloaded to a separate service in a lower-level language if Node proves too slow for heavy text extraction, but try to use Node libraries or cloud APIs first to keep things simple.
- *Podcasts (audio):* You'll likely use podcast RSS feeds to get episode metadata and audio file URLs. For transcription, using an external API (like OpenAI Whisper API or AssemblyAI) might be easiest to start – you could send the audio and get text back. This adds cost, so as a fallback consider running an open-source transcription locally. If doing local transcription at scale, implementing that in a faster language (Rust or Go) or using a GPU-enabled service would be necessary, since pure TypeScript isn't ideal for heavy CPU tasks. A pragmatic approach: initially use a paid API with reasonable free tier (OpenAI Whisper's pricing is low per minute), and monitor usage. If it exceeds budget, move to a self-hosted Whisper server (there are Rust bindings for Whisper or C++

implementations). Keep this as a modular component (e.g. the Node worker puts audio files in a queue for a Rust service to transcribe).

- *Articles/Blogs:* Most will be accessible via RSS or site APIs. The engine can fetch the RSS feed, then fetch each new article URL and extract content (use an HTML parser or readability library in Node to extract text). To accelerate this, you could utilize an open-source **RSS aggregator service** (discussed later) so that you don't individually fetch thousands of feeds. If HTML parsing in Node is slow or memory-heavy for huge pages, consider Go (with Goose or similar) or using existing hosted services for text extraction (like Diffbot, but that's not cheap; better to stick with open source).
- *Social Media:* If targeting Twitter/X, their official API is no longer freely accessible for large volume. You might skip direct Twitter ingestion due to cost, or use unofficial alternatives. For Reddit, you can use their RSS feeds for subreddits or an API wrapper. For other platforms, consider community-driven APIs or RSS feeds (some platforms or third-party services provide RSS for user timelines).

Using **TypeScript/Node** for orchestration and I/O is a priority, but be pragmatic – if certain tasks are fundamentally unsuited to Node (CPU-bound jobs like video encoding or large PDF parsing), you can write a microservice in Go or Rust to handle those. The architecture can remain the same (jobs in the queue trigger a request to the specialized service). For example, a Rust service could watch a Redis queue for "transcription jobs", process an audio file to text, and then the Node engine picks up the result. However, **avoid premature optimization**: start with Node-friendly approaches and only introduce Go/Rust components if monitoring shows Node can't keep up or memory usage is too high.

Real-World Example – Midday's Engine: Midday (open-source business automation tool) built a similar background processing engine entirely in TypeScript. They use Supabase (Postgres) for storage, and **Trigger.dev for background jobs** (embedding generation, document parsing) [11]. The result is an engine that processes **thousands of tasks daily** reliably, achieving 99.9% uptime with robust error handling [13] [14]. They scaled to 11,500 customers with a two-person team by leveraging serverless and background jobs effectively [15] [16]. Midday's architecture proves that a well-designed TS-first pipeline can handle high volume. We can mirror their approach by: using a Postgres DB (or Supabase) for state, Node workers for tasks, and a service like Trigger.dev for scheduling, **resulting in a scalable system that remains simple to maintain** [17] [2].

API Layer: Separating Engine from User-Facing API

To keep things performant and maintainable, **decouple the ingestion engine from the API** that clients interact with. This ensures that heavy lifting (downloads, parsing, ML processing) doesn't bog down request/response times. The API's job is only to serve already-processed data (and possibly trigger new ingestions via lightweight signals or enqueueing, if a user manually submits a source).

- **Cloudflare Workers for API:** Cloudflare Workers are a great choice for a globally distributed, low-latency API. They excel at handling lots of concurrent lightweight requests with very low overhead. You can deploy a small API (for example, an endpoint to search or fetch ingested content) to Workers and benefit from built-in scaling and CDN caching. Since Workers run on V8 isolates, you write your API in TypeScript (using the Worker runtime APIs or frameworks like Cloudflare **Pages Functions** which support a Next.js-like API layer). Workers have a 50ms CPU time limit per request (which is fine for simple DB queries or cache lookups) and can respond in under 100ms globally. This means your users get very fast responses. And importantly, the ingestion engine (running elsewhere) can push data to a database that the Worker reads. Cloudflare's pricing is usage-based and extremely cheap for the scale of API calls (millions of requests would still be in the few dollars range).

- *Connecting to the DB:* A Cloudflare Worker can directly call external APIs, but it can't directly maintain a persistent database connection. The typical pattern is to expose a **HTTP endpoint** for data queries. For example, you might create a minimal API (in front of your database) or use something like Supabase or Hasura GraphQL as an interface. Supabase, if used, has a REST and GraphQL API layer for the Postgres database – the Worker could call Supabase's REST endpoints (with an API key) to fetch data. Alternatively, use Cloudflare D1 (a lightweight SQLite database at edge) to cache or store smaller data for ultra-fast reads, but for 10k+ daily items, a central Postgres is likely better.
- If Cloudflare Workers' model doesn't fit a certain need (e.g. server-sent events or long-lived requests), another good option is to run the API on a lightweight Node server on Fly.io or a similar provider. Midday, for instance, hosts their API (tRPC/Next.js backend) on Fly.io, separate from the frontend on Vercel [18]. For Zeek, you could run an Express/Fastify or tRPC server in Node that connects to the DB. This can be a single small VM ($5–$10/mo range) because the API load is mainly I/O bound. The key is it's separate from the engine process – so even if the engine is busy or down, the API can still serve data (perhaps stale data if updates paused, but not completely dead).
- **Frontend/UI:** If Zeek has a user-facing interface or dashboard, host it separately (e.g. a Next.js app on Vercel or Netlify, or a static React app on Cloudflare Pages). The front-end calls the API service for data. This separation (front-end vs API vs engine) aligns with the principle of isolating heavy background work. The UI should never directly perform ingestion; at most it might send a request to the API to add a new source or trigger a refresh, which the API can handle by enqueueing a job in the engine (for example, the API could publish a message to a queue or insert a row in a "pending jobs" table that the engine watches).
- **Optimizations:** The API can use caching or batching for read requests. For example, Cloudflare Workers have a Key-Value store and D1 DB – you might cache frequent queries (like "latest 10 articles ingested") in KV for a few minutes to reduce load on your primary DB. Also consider GraphQL with an in-memory cache or Apollo's CDN caching for repeated queries. Given the modest budget, focus on simple caching strategies that reduce database calls (like HTTP `ETag` / `Last-Modified` headers and Cloudflare's cache for static responses or rarely changing content).

In summary, **keep the API stateless and as thin as possible**. It should mostly be an interface to query the database (and possibly the search index). All the complex logic happens in the engine. This way, scaling the ingestion (which might require CPU and memory) does not affect the scaling of the API (which needs to handle bursty traffic). This division is a common pattern in high-volume systems – for example, Midday's "Engine API" (running on Fly) handles bank connections and communicates with their background jobs, separate from the Next.js frontend [19]. By following this pattern, Zeek's API will remain responsive even under heavy ingestion load.

Reliability and Observability

With thousands of jobs and multi-step pipelines, **observability** is crucial. We want to detect failures early, understand performance bottlenecks, and ensure the system self-recovers when possible. Here's how to build reliability and monitoring into the architecture:

- **Structured Logging:** Use a logging library in Node (like Winston or Pino) to output structured logs (JSON logs with context: source, job ID, timing, errors, etc.). Aggregating logs from multiple services (engine workers, API service) is easier if they all emit to a central sink. A cost-effective stack is

Grafana Loki for log aggregation (Loki is like a Prometheus for logs). You can run Loki + Grafana on a small VM or use Grafana Cloud's free tier. Logs can be pushed from your services to Loki (for example, via a Promtail agent or directly via HTTP API). This allows querying all logs in one place and setting up alerts (e.g. alert if many errors in a short time). For simplicity, you might also log to a file and use a file shipping solution, but a cloud approach might be simpler given the distributed nature of engine and API.

- **Metrics (Monitoring):** Expose metrics from both the engine and API. There are Node libraries like **Prometheus client (prom-client)** that let you record custom metrics (counters, gauges, histograms). For example, you can track "jobs_processed_count{status=success}" or "job_duration_seconds" and "api_request_count". Host a **Prometheus** server (or use a hosted one) to scrape these metrics periodically. Prometheus + Grafana will allow you to graph trends (jobs per minute, average processing time, memory usage if exported, etc.). Set up **Grafana dashboards** to visualize key stats (ingestion rate, backlog queue length, error rates). Grafana can also tie into alerting (send email/Slack if certain thresholds exceeded, like queue length too high or no jobs processed in X minutes).
- **Distributed Tracing:** For a sophisticated view, implement tracing with **OpenTelemetry** in your Node app. This can trace a job from fetch to processing to DB insert, or an API request through to DB query. The traces can be exported to an open-source tracer like **Jaeger** or Grafana **Tempo** [20]. Tracing is helpful if you have complex workflows (like a job that triggers sub-tasks) to pinpoint bottlenecks. However, it might be overkill initially. You can start with logs + metrics which cover most needs. If using Trigger.dev, note that it already provides a level of tracing/monitoring for task flows in its UI (each job's steps and timing).
- **Automatic Alerts & Recovery:** Set up automated alerts for critical conditions: e.g., if a job fails after retries, have the orchestrator send a notification (Trigger.dev can integrate with Slack or email for alerting on failures [2]). If using a queue like BullMQ, consider a failure handler that moves dead jobs to a "dead-letter" queue and notifies you. Ensure the engine services are run under a process manager that restarts them on crash (or use Kubernetes/a container service that auto-restarts containers). With 10k tasks/day, you'll inevitably hit some failures (network issues, bad data) – **the system should log and recover without manual intervention** for most cases. Manual attention should only be needed for truly unexpected errors or consistently failing sources.

By investing in observability from day one, you create a feedback loop: you can see how long each source takes to ingest, which sources fail often, how close you are to resource limits, etc. This guides optimizations. For example, if logs show a particular RSS feed often times out, you might remove or replace it. If metrics show memory spikes, you might increase instance size or refactor that job's code.

Grafana Integration: All the above can be tied into a Grafana dashboard system. Grafana can display logs (via Loki), metrics (via Prometheus), and even traces (via Tempo) in one place. This is convenient for a small team. The stack is open-source and Grafana has guides on combining these (the **"PLG stack"**: Prometheus, Loki, Grafana) for full observability. By using Grafana's alerting, you can get pager-duty style notifications when ingestion falls behind or errors spike. Midday's team, for instance, mentioned having "comprehensive logging and performance tracking with automatic alerting" as part of their engine [21] – adopting a similar stance will ensure Zeek runs reliably 24/7.

Ingestion Sources & External Services

To ingest a wide breadth of content efficiently, **leverage existing APIs and open datasets** where possible. This saves development time and can reduce the load on your system (since you can retrieve aggregated data instead of scraping everything yourself). Below are some resources for the types of sources mentioned:

- **RSS Aggregators:** Rather than individually polling 10,000 RSS feeds, you can use a feed aggregation service or host an aggregator. Open-source tools like **Miniflux** or **FreshRSS** can be self-hosted – they will fetch feeds for you and store new items in a database. Miniflux, for example, is a lightweight RSS reader API you can run (written in Go). You could periodically query Miniflux via its API to get all new items across feeds, instead of hitting each feed yourself. This centralizes feed fetching and ensures robust parsing (Miniflux handles various feed formats). If self-hosting is undesired, a service like Feedly has an API (though not free at scale) or there's **NewsAPI** for news articles (paid beyond a point). For a budget-conscious approach, a single $5 VPS running Miniflux could cycle through thousands of feeds; your engine just pulls from Miniflux's database of new posts.
- **ArXiv & Scholarly Articles: arXiv** offers open APIs for its papers. It has RSS feeds for each subject category (updated daily with new papers) and an OAI-PMH interface for bulk harvesting [22] [23]. You can subscribe to relevant arXiv feeds and ingest new papers' metadata easily (just parse the XML). ArXiv's API is free and open as long as you acknowledge their data [22]. For broader scholarly data, consider the **Semantic Scholar Open API**. Semantic Scholar provides a comprehensive database of papers, authors, and citations. Their REST API lets you query papers by various IDs, get citations, etc., and **no API key is required for many endpoints** (they allow ~1000 requests per second for public use) [24] [25]. This is a rich source for scientific content if Zeek needs to ingest academic knowledge. Another resource is **OpenAlex** (an open catalog of scholarly papers with an API similar to Semantic Scholar).
- **Podcast Index:** The **PodcastIndex.org** is an open project that aggregates podcast feeds globally. They provide a free API for searching podcasts and fetching podcast feed info [26] [27]. Instead of crawling thousands of individual podcast RSS feeds, you can use Podcast Index to discover feeds and even get lists of recent episodes across the whole index. For example, they have an endpoint for "recently added episodes" which could let you ingest fresh podcast episodes across the board without checking each feed [28]. You will need to sign up for a free API key and include authentication headers, but it's free for reasonable use [29]. Using Podcast Index can **accelerate ingestion breadth** by tapping into their database of podcasts/episodes [26].
- **Social Media:** This is tricky in 2025 due to API restrictions. For Twitter (X), the free API is extremely limited. If social media ingestion is a priority, consider alternatives:
- **RSS-Bridge or RSSHub:** These open-source projects provide unofficial RSS feeds for social platforms. For example, RSSHub can generate an RSS feed for a Twitter user or a Facebook page without using official APIs. Deploying RSSHub (Node.js) and adding bridges for the needed platforms could give you a feed of posts to ingest. Keep an eye on the legality and reliability, as these scrape the sites.
- **Pushshift (Reddit):** Reddit offers free access to public data through Pushshift or their own API. Reddit has RSS for subreddit new posts as well. For other forums or aggregators (HackerNews, etc.), many have RSS feeds for new stories which you can tie into the same RSS ingestion mechanism.

- **Mastodon/Bluesky/Nostr:** If your goal is to ingest "social content" broadly, perhaps focusing on decentralized platforms might be easier since they have open APIs. Mastodon servers provide

ActivityPub APIs and many have public timelines. Nostr is an open protocol with many relays you could tap into if needed. However, these might be beyond scope if the focus is mainstream content.

- **YouTube and Video Transcripts:** While not strictly open, the YouTube Data API is the official way to get video info. It can retrieve captions if they are uploaded (with `captions.list` endpoint) – though in many cases you might need to use a tool to fetch auto-generated captions. There are open-source libraries (like `youtube-transcript-api` in Python) – since we avoid Python, you might call a subprocess for such a script or use a Node library or headless browser approach to fetch the transcript. Also consider using the **Invidious** API (Invidious is an open-source front-end for YouTube that has an API which can return captions and video info without API keys). You could deploy an Invidious instance or use a public one to fetch transcripts in a pinch.

Using these external resources can dramatically increase how much content Zeek can cover with minimal effort. For instance, by integrating with PodcastIndex and arXiv, you instantly cover thousands of podcasts and scientific papers without custom scraping. Just be mindful of **rate limits** and terms of use: - PodcastIndex might limit requests per day (their free tier is generous but check current limits). - ArXiv asks for at most one request per second for their API to be polite [30] – which is fine if you only fetch daily updates. - Semantic Scholar's open API is rate-limited (1 request/sec with an API key, or shared pool without key) [31] , so you'd want to cache results or use their dataset dump for massive ingestion.

Cost Efficiency and Deployment Considerations

All of the above must be achieved under **<$500/month**. This budget is ample if using a judicious mix of managed services and self-hosting:

- **Compute Infrastructure:** Aim to run the ingestion engine on cost-effective cloud instances. A single `$20–$40/mo` VM (e.g., a 4 vCPU / 8GB RAM droplet on DigitalOcean or an equivalent on Hetzner) can likely handle 10k jobs/day in Node if written efficiently. Node can spawn dozens of concurrent async operations (network calls) with low memory, so the bottleneck will be CPU if heavy parsing or external API latency. If one instance isn't enough, you can have two smaller instances (one fetching, one processing, etc.). Using Docker with a compose or a lightweight Kubernetes (e.g. k3s on a VM) could help orchestrate if you split roles, but that might be overkill. Simpler: use a **Process Manager** like PM2 or Docker Compose to run the worker processes and ensure they restart on crashes. This avoids Vercel's limits and gives full control over long jobs.
- **Database: Supabase** is a convenient choice – their hosted Postgres can handle moderate workloads and they have a free tier and inexpensive plans. Supabase at $25/month can handle millions of rows and has 50GB of storage, which might be enough for textual data of 10k items/day for some time. It also provides storage (for storing audio files or images if needed) and auth (if you need user accounts). Another option is a managed Postgres from Render or DigitalOcean (~$15-20/mo for basic). If you're comfortable managing a DB yourself, a $10 VM with Postgres could work but requires maintenance. Given budget, a managed DB is worth it for reliability. Ensure you index the tables properly (e.g., full-text index if searching titles, etc.) to keep query performance high.
- **Search Index:** If you need full-text or vector search over the content, consider **Typesense** (open-source, easy to self-host, built in Rust for speed). Midday uses Typesense for search in their app [32] . Typesense can run on a $5-10 VM for moderate data sizes and provides instant search results. Another choice is **Meilisearch** (similar idea, also lightweight). Both are free to use. If vector search (for semantic similarity) is required, you can use Postgres with pgvector (as Midday did [11]) to avoid

running separate vector DBs – this keeps cost down by leveraging the existing DB. Only add a dedicated search engine if needed for speed or complex querying.

- **Queue/Cache:** If using Redis (for BullMQ or caching), you can run Redis on the same box as the engine (to save cost) or use a small managed Redis (Upstash offers a free/cheap serverless Redis that might suffice). Redis's memory needs will depend on how many jobs and their payload size – 10k jobs/day with small payloads is trivial (a few MB of memory). So a $5 instance or free tier should handle it.
- **Cloudflare Workers:** The API layer on Cloudflare Workers could likely fit in the Workers free plan (which includes 100k requests/day). If you exceed that, the pricing is still low (e.g. $0.50 per million requests). So essentially $0 or a few dollars for the API. If you use Workers KV or Durable Objects, there might be slight costs, but likely negligible at our usage. Alternatively, if an API server on Fly.io is used, a single shared-cpu VM is around $5-10/mo.
- **External API costs:** Most of the mentioned external sources are free or open. PodcastIndex is free (community funded). Semantic Scholar is free. ArXiv is free. For transcription of podcasts, if you use OpenAI Whisper API, it's ~$0.006 per minute – if you transcribed, say, 100 hours of audio per day (which is *a lot*, 100 hours = 6000 minutes, which would be $36), you'd spend $1000+ in a month just on that, so you wouldn't do that at large scale on a $500 budget. More realistically, you might select a subset of podcasts or use shorter summaries. It might be wise to not transcribe every podcast fully – maybe pull episode descriptions (from RSS) or use PodcastIndex's transcripts if available (some podcasts provide transcripts). If transcription is crucial, consider running Whisper locally on a GPU you own or rent a GPU instance for cheaper (some providers offer GPU hourly rates that might be cheaper if usage is high).
- **Observability stack:** Grafana, Prometheus, Loki can all be self-hosted on one VM (they don't require huge resources for moderate data volumes). There are also hosted options: Grafana Cloud has a free plan that includes some metrics and logs – that could be enough to start. If you self-host, maybe dedicate another $5-10 VM for it, or run it on the same machine as something else (just be careful with resource contention). Alerts could be sent via free services (Slack, email, etc., which Grafana supports out of the box).

Overall, this architecture mostly relies on **open-source software** and modest cloud instances. You avoid heavy managed services (no need for Kafka or expensive serverless functions). Everything suggested (Node, Redis, Postgres, Workers) has a low footprint. **$500/month** can easily cover a few VMs, a managed DB, and some API costs with margin to spare. For example, approximate breakdown: $40 engine server + $25 DB + $10 search server + $10 for misc (Redis or storage) + $10 Grafana server + $5 domain = ~$100. Even if we scale up instances or have multiple, it should remain well under $500. This leaves room if you need to occasionally scale up (e.g. during a spike you might enable a second worker server or use a cloud function burst – those costs would be incremental).

Critically, the design emphasizes **simplicity and maintainability**. All components are replaceable and loosely coupled: if one tool doesn't work out, you can swap it (e.g. switch from Trigger.dev to BullMQ, or from Cloudflare to a Node API) without a complete rewrite. Start with straightforward implementations and only optimize when metrics show a need. By standing on the shoulders of open-source solutions (for jobs, feeds, search, etc.), you avoid reinventing wheels prematurely. This modular, monitored system will set a strong foundation for Zeek's high-volume ingestion needs.

Recommended Tools and Services (By Category)

Finally, here's a summary list of recommended technologies grouped by category for quick reference:

- **Language & Frameworks:**
- **TypeScript/Node.js** – Primary choice for writing the ingestion engine and API code. Leverage Node frameworks as needed (e.g. Fastify or tRPC for the API).

- *Fallback:* **Go or Rust** for specific performance-critical microservices (e.g. heavy CPU tasks or faster feed parsing), if Node proves insufficient.

- **Job Orchestration & Scheduling:**

- **Trigger.dev** – Easiest way to build reliable background jobs in TS with scheduling, retries, and a built-in UI [1] [2].
- **BullMQ (Redis)** – Robust queue for Node with support for thousands of jobs/sec and features like delayed jobs and rate limiting [4] [33]. Use with **Bull Board** or **Arena** for a web dashboard.
- **Cron/CronJobs:** Use library like Node-cron for simple scheduled tasks, or Trigger.dev's scheduling, or even OS cron plus API calls to kick off jobs (for very simple scheduling needs).

- *Alternative:* **Temporal (Temporal.io)** – If workflows get complex, provides an open-source orchestration engine with a TypeScript SDK and **Web UI for monitoring** [7] [34] (heavy to self-host, consider only if needed).

- **Data Storage & Search:**

- **PostgreSQL (Supabase)** – Main database for storing ingested content and metadata. Supabase provides an easy Postgres with auth and REST API out-of-the-box [35]. Enable **pgvector** if vector similarity search is needed [11].
- **Typesense** – Open-source search engine for full-text search, typo-tolerant queries on articles, etc. Very fast and simple to deploy (binary or Docker). Use if you need advanced search beyond PostgreSQL's capabilities.
- **Redis** – In-memory datastore for queue (if using BullMQ) and caching frequently accessed data. A small instance will do; can also use Redis for a simple rate-limiter counter.

- **Object Storage:** If storing large files (audio, video or PDFs), use something like Supabase Storage or AWS S3 (Wasabi or Backblaze B2 are cheaper alternatives) – typically pennies per GB.

- **Compute / Hosting:**

- **Cloudflare Workers** – Deploy the API endpoint globally with minimal cost. Great for handling many small requests quickly.
- **Fly.io / DigitalOcean / Hetzner Cloud** – for running the Node ingestion engine and any other long-running services (Redis, Postgres if self-hosting, etc.). These give you full control over environment for the TypeScript engine and are affordable. For instance, Fly can deploy Docker containers close to your location with autoscaling if needed.

- **Docker** – Use Docker to containerize components, making it easy to deploy and scale (e.g., one container for the engine worker, one for Typesense, etc., orchestrated via docker-compose on a single VM or multiple).

- **Supabase** – (if used) hosts the Postgres and offers serverless Edge Functions (though Edge Functions have 60s timeout, not for long jobs) and an authentication layer if needed for users. Supabase's free/pro tiers can offload some backend work (like you can use Row Level Security and policies to let the API directly query Postgres via Supabase safely).

- **Observability & DevOps:**

- **Grafana** – Dashboard UI to visualize metrics and logs in one place. Can run locally or use Grafana Cloud (free tier).
- **Prometheus** – Metrics collection. Run a Prometheus server to scrape metrics from Node services (exposed via an HTTP `/metrics` endpoint by `prom-client`). Create alerts (via Alertmanager or directly in Grafana) for key metrics (error count, queue length).
- **Grafana Loki** – Log aggregation database. You can push logs from the engine/API to Loki and use Grafana to search them (similar to how you'd use ELK stack but much lighter weight).
- **Sentry (optional)** – For error tracking on the API and maybe engine. Sentry's free tier could catch exceptions with stack traces, which is useful for debugging. Not Grafana-based, but it's a quick way to get alerting on exceptions in code.

- **Slack/Webhooks** – Integrate alerts to Slack or email. For example, Trigger.dev can post job failures to Slack, or Grafana can send webhook alerts. This ensures you're notified quickly of issues.

- **External Data Sources (to expand ingestion):**

- **RSS Hub / Aggregators:** *Miniflux* (open-source RSS aggregator with API) to manage feed polling; *FreshRSS* if you need a UI as well. If not self-hosting, consider *Feedly API* (paid) or at least use it to bootstrap a list of popular feeds. **RSSHub** (open-source) to generate RSS feeds for sites that don't have them (e.g. YouTube, Twitter, Instagram).
- **Podcast Index API** – Community-driven index of podcasts, free API to search and get feed info [26]. Use it to discover and update podcast episodes without crawling thousands of feeds.
- **arXiv API & OAI-PMH** – Grab new scientific papers daily from arXiv's feeds or OAI interface (free, just follow their rate limit guidance) [22].
- **Semantic Scholar API** – Access metadata for scholarly articles (papers, authors, citations) with a free, high-rate API [25]. Useful for ingesting research content and linking it semantically.
- **Social media** – *Reddit:* use subreddit RSS or Pushshift; *Twitter:* consider Nitter or scraping via RSSHub (if within legal use); *Mastodon:* each instance provides Atom feeds for users or hashtags which can be ingested.
- **YouTube Data API** – Official API to get video details and captions when available. Requires API key (free tier allows a significant number of reads daily). For transcripts of videos without official captions, consider alternate strategies (as discussed in engine section).
- **Others:** *News APIs* (e.g. NewsAPI.org for news articles – free tier 500 requests/day), *Wikipedia dumps or RSS for new pages*, *Library feeds (e.g. via Open Library APIs)* – depending on Zeek's focus, there are many free datasets and APIs for content.

By thoughtfully combining these tools, Zeek’s ingestion engine will be **scalable, observable, and maintainable**. We emphasized using TypeScript for most components to leverage the team’s strength and keep a unified codebase, resorting to Go/Rust only when truly necessary. This architecture separates concerns (ingestion vs serving) and uses battle-tested solutions for the hard parts (queueing, DB, search, etc.), ensuring that even with 10k+ sources per day, the system remains **reliable** and within budget. With monitoring in place and a bias for simplicity, you can iteratively improve performance as load grows, without costly redesigns or firefighting.

---

[1] [11] [13] [14] [21] Building an Automatic Reconciliation Engine: How We Match Receipts to Transactions | Midday
https://midday.ai/updates/automatic-reconciliation-engine/

[2] [3] [9] [15] [16] [17] [19] Midday's Automated Bank Synchronization, powered by Trigger.dev | Trigger.dev
https://trigger.dev/customers/midday-customer-story

[4] [5] [6] [10] [12] [33] BullMQ - Background Jobs processing and message queue for NodeJS | BullMQ
https://bullmq.io/

[7] Temporal : Revolutionizing Workflow Orchestration in Microservices ...
https://medium.com/@surajsub_68985/temporal-revolutionizing-workflow-orchestration-in-microservices-architectures-f8265afa4dc0

[8] [34] Temporal Web UI | Temporal Platform Documentation
https://docs.temporal.io/web-ui

[18] [32] [35] GitHub - midday-ai/midday: Invoicing, Time tracking, File reconciliation, Storage, Financial Overview & your own Assistant made for Freelancers
https://github.com/midday-ai/midday

[20] Grafana Tempo OSS | Distributed tracing backend
https://grafana.com/oss/tempo/

[22] [23] arXiv API Access - arXiv info
https://info.arxiv.org/help/api/index.html

[24] [25] [31] Semantic Scholar Academic Graph API | Semantic Scholar
https://www.semanticscholar.org/product/api

[26] [27] [28] [29] PodcastIndex - Free Public APIs
https://openpublicapis.com/api/podcastindex

[30] An error occurred saving with arXiv.org. Attempting to save using ...
https://forums.zotero.org/discussion/115157/an-error-occurred-saving-with-arxiv-org-attempting-to-save-using-save-as-webpage-instead