



Develop. Preview. Ship.

VERCEL OBSERVABILITY LIKE LOGGING SYSTEM
SYSTEM DESIGN CAPSTONE PROJECT

Google Drive Link: https://drive.google.com/drive/folders/1GQ_HEIZjUQ4GP7l5kzbgyWioWx6bNi1i?usp=sharing

SYSTEM DESIGN CAPSTONE PROJECT

I named my system:

Log Stream

A Distributed Logging Platform for Deployment Systems

Ingestion

Streaming

Batch Processing

Storage at Scale

PRESENTER

Ammaar Naveed

REFERENCE SYSTEM

Vercel Observability / Drains

Table of Contents

01

Context & Problem

Slides 4–6

02

Requirements

Slides 7–8

03

Scale & Capacity Planning

Slides 9–10

04

Data Model & API Design

Slides 11–13

05

Ingest Path Architecture

Slides 14–20

06

Storage Engine & Tiering

Slides 21–23

07

Serve Path Architecture

Slides 24–27

08

Five-Year Capacity Projection

Slide 28

09

Conclusion

Slide 29

Vercel, The AI-First Cloud To Ship Applications Faster

A deployment platform that takes developers from source code to a live, globally-served application without managing servers, CDNs, or scaling by hand.

Build pipeline	Isolated containers build every push — preview per branch, production on merge.
Edge delivery	Anycast-routed global CDN serves the artifact, with instant rollback.
Observability	Build logs, runtime logs, traces, analytics — the subsystem this deck designs.

git push → build → deploy

The core loop this platform automates end to end — the three pillars at left are what make it reliable at scale.

PROBLEM

Why Logging Is A separate, Hard Problem

01

Write volume exceeds the app

Every build, function call, and edge request generates log lines. Log write throughput exceeds the primary application's own request volume.

02

Two incompatible access patterns

Developers need logs the instant they're written. Teams also need months of history aggregated for trends.

03

Backpressure risk

If storage slows, a producer drops logs or blocks. Neither is acceptable without a buffer between the two.

WHY IT DESERVES ITS OWN SYSTEM

Logging is a cross-cutting concern, every component in the platform is a source. Solved once, centrally, it's simpler than solving it piecemeal in every subsystem.

SCOPE

Defining Scope

IN SCOPE

- Log ingestion from millions of sources
- Durable buffering & backpressure handling
- Real-time streaming (live tail) to viewers
- Stream processing: windowed aggregation
- Storage engine & hot/warm/cold tiering
- Batch export pipeline for long-term analytics
- Query & retrieval patterns

OUT OF SCOPE

- The build pipeline itself
- Request routing / edge CDN
- Rollback mechanism
- Rate limiting on the deploy API
- Alerting rules / notification delivery
- Full dashboard / query UI

What System Must Do

01 Accept log writes from millions of concurrent sources

02 Never lose a log line once a write is acknowledged

03 Preserve per-source ordering of log lines

04 Stream new log lines to live viewers within ~1 second

05 Support fan-out: multiple viewers tailing the same source

06 Aggregate logs into real-time metrics (error rate, volume)

07 Support time-range and full-text query over history

08 Tier data automatically: hot → warm → cold, by age

09 Export logs/traces to external destinations (drains)

10 Enforce per-tenant isolation across all reads and writes

How Well System Must Perform

Ingestion latency

Single-digit ms
to acknowledge

Streaming latency

< 1s
to live viewers

Durability

Zero loss
once acknowledged

Throughput

Tens of thousands
lines / sec, peak

Query latency

Sub-second (hot)
Seconds (cold)

Cost efficiency

Cost/GB drops
as data ages

Isolation

No cross-tenant
degradation

Ordering

Preserved under
retries / replay

SCALE ASSUMPTIONS

Log Volume

Approx. 10M platform DAU, 15M builds/day, 90,000/sec peak edge requests.

150

avg. log lines per build

2.25B

build log lines / day

90,000/s

edge request events, peak

~10B

total log lines / day

EXPLANATION

Edge requests alone (~90,000/sec peak) generate over 2,500× more log events per second. Log volume exceeds the primary application.

Ingestion Throughput

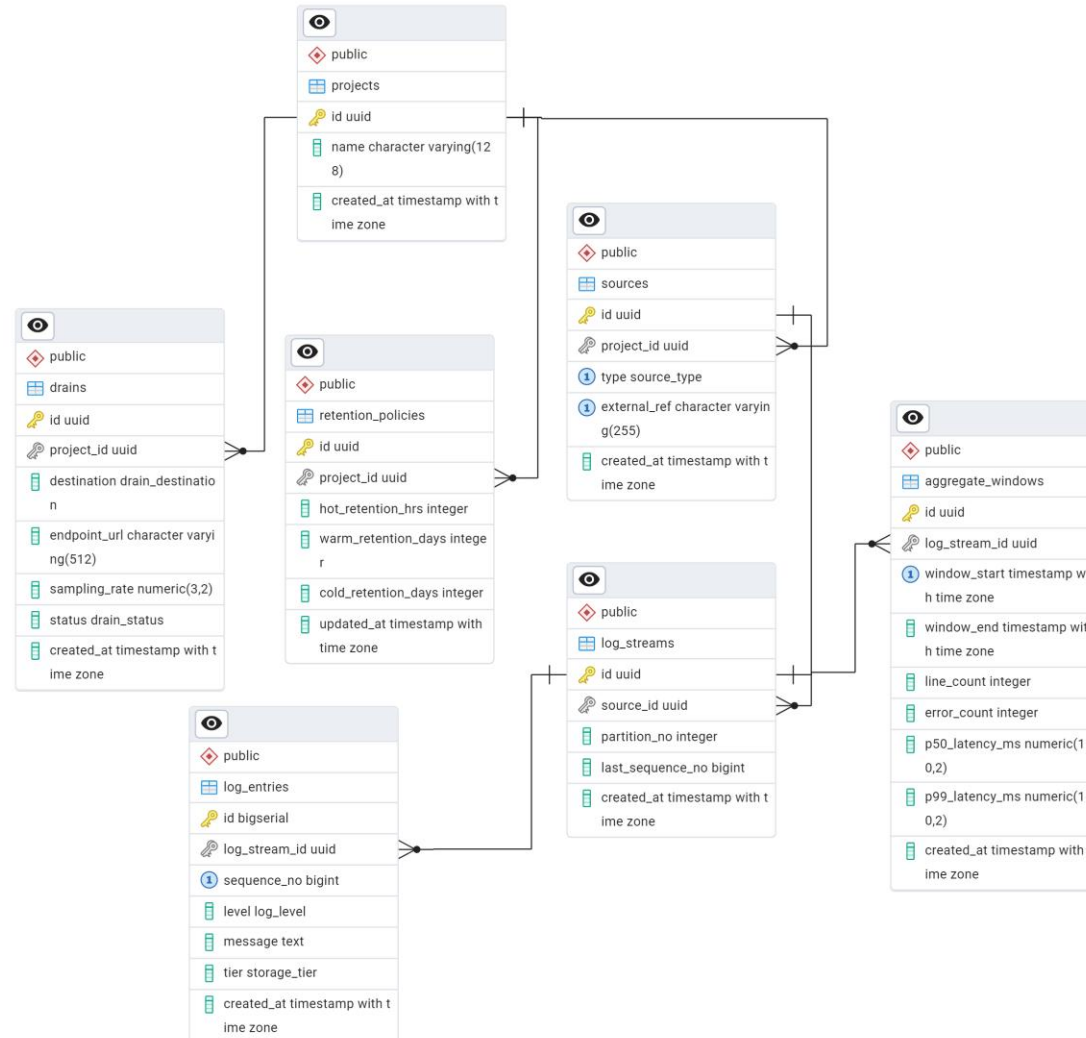
METRIC	VALUE	NOTE
Avg. log line size	~180 bytes	Timestamp + level + source ID + message
Peak log events/sec	~90,000/sec	Dominated by edge/runtime logs
Peak ingest bandwidth	~16.2 MB/sec	$90,000 \times 180 \text{ bytes}$
Daily ingest volume	~1.4 TB/day raw	$16.2 \text{ MB/sec} \times 86,400\text{s}$
After columnar compression	~140 GB/day	~10× — see Slide 13

Core entities



-
- Source (build / function / edge node): Generates log entries
 - LogStream: One ordered sequence of logs, per Source.
 - LogEntry: A single log line inside a LogStream.
 - AggregatedWindow: A 1-minute summary (rollup) of a LogStream.
 - Drain: An external destination logs get exported to.
 - RetentionPolicy: Defines how long logs stay in each storage tier.

Entity Relationship Diagram



REST & WebSocket Endpoints

POST	/v1/logs	Ingest a batch of log lines (producer write path)
GET	/v1/logs/{sourceId}	Time-range query over a source's historical logs
GET	/v1/logs/{sourceId}/tail	Open a live-tail stream (upgrades to WebSocket)
GET	/v1/projects/{projectId}/aggregate	Windowed metrics: error rate, volume, latency
POST	/v1/drains	Register an external export destination
GET	/v1/drains/{drainId}/status	Check delivery health of a configured drain

WebSocket: GET /v1/logs/{sourceId}/tail upgrades to a persistent connection. Server pushes { type: "LOG_LINE", sourceId, sequenceNo, ts, level, message } per new line.

Ingest Path

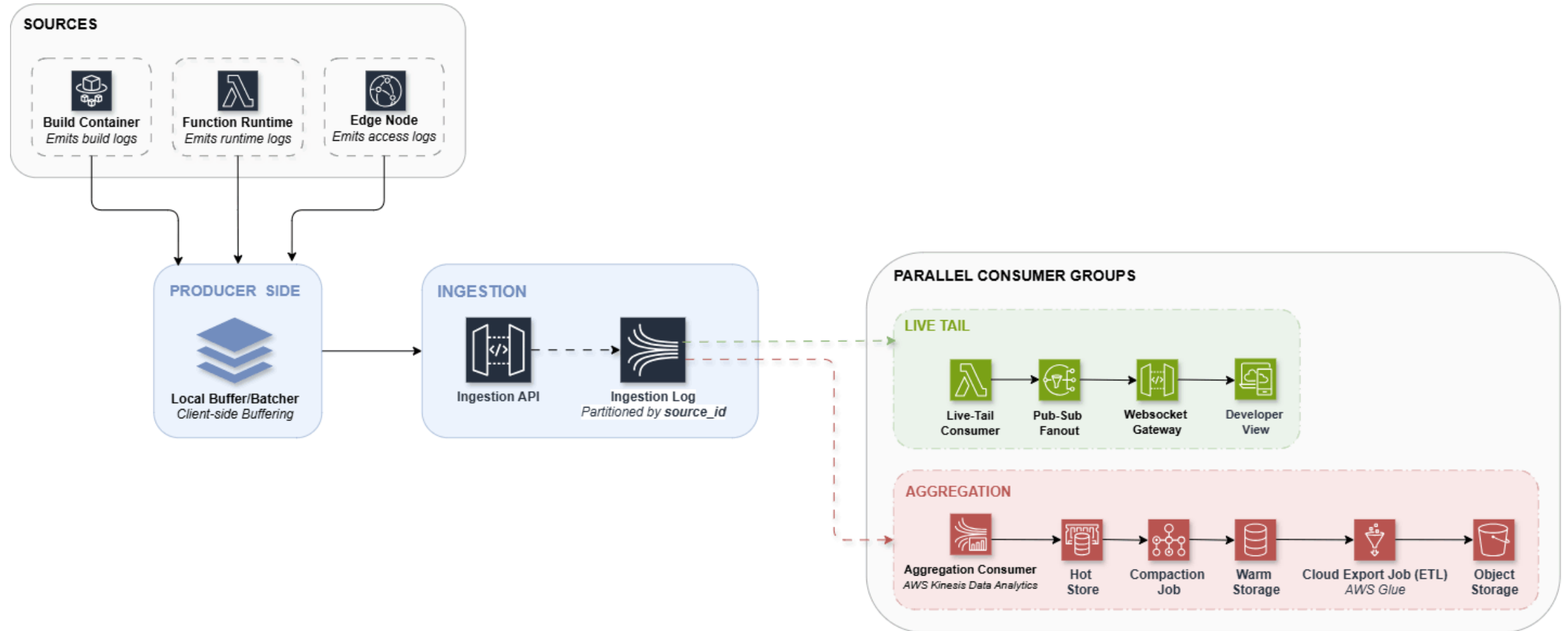
SOURCES —————> STORAGE

Ingestion

Producer

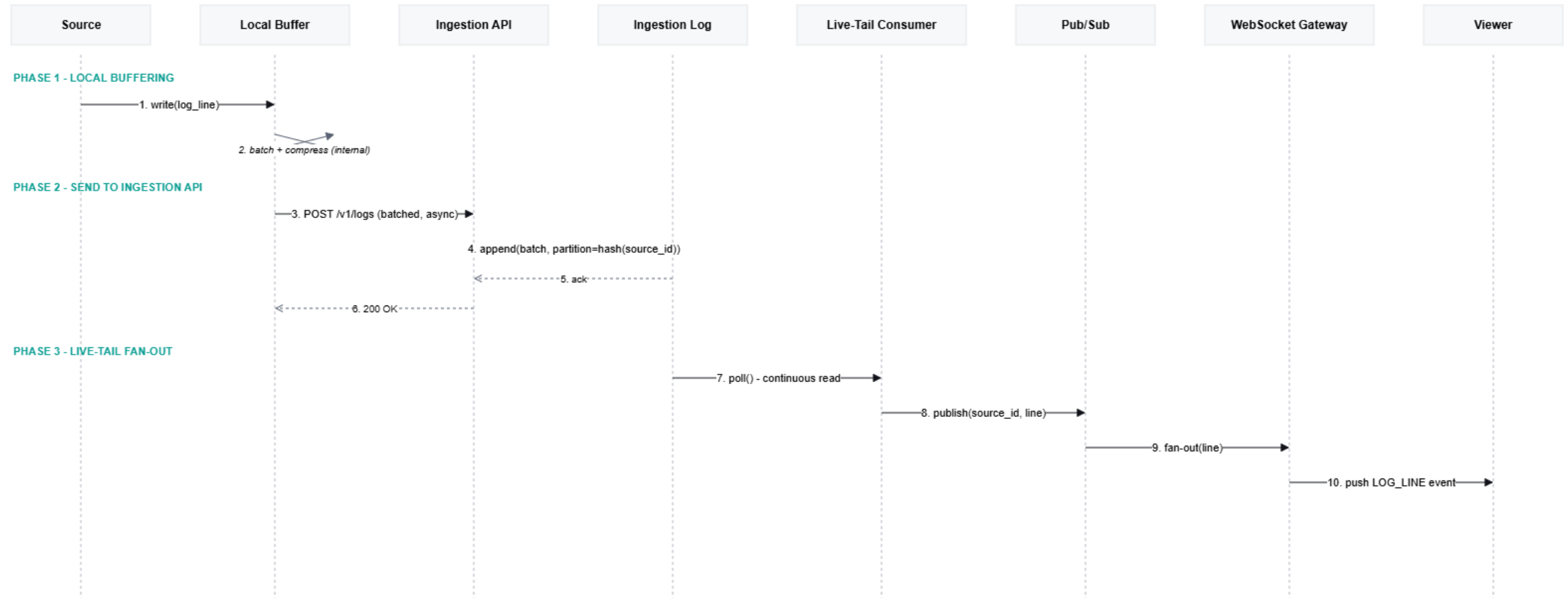
Distribution

Ingest Path Diagram



SEQUENCE DIAGRAM

Ingest + Live Tail Sequence



The viewer sees the new log line in under 1 second. No aggregation or storage write sits on this path.

Producer Side (Buffering & Backpressure)

- | | | |
|-----------|-----------------------------|--|
| 01 | Local Buffering | Every source (build container, function runtime, edge node) buffers log lines locally before sending. Producer's real work is never blocked by slow network. |
| 02 | Client-side Batching | Lines are batched and compressed before sending. |
| 03 | Async Processing | The producer never waits on ingestion acknowledgment to continue. |
| 04 | Bounded Queue | If ingestion is unreachable too long, the local buffer caps out. Oldest lines drop first, newest are kept. |

Distributed Commit Log (Durable Log)

Between millions of producers and slower consumers, traditional queue is never helpful.

Durability

Every write is persisted to disk and replicated across brokers before acknowledgment. Data never loses when consumer is down.

Replay

A failed batch processor can restart and reprocess from where it left off.

Decoupling

Fast producers are never blocked by slow storage processing at consumer side.

Horizontal scale

Adding capacity leads to adding partitions and brokers. There is no architectural ceiling as source count grows.

At approx. 16.2 MB/sec peak, a consumer glitch without this buffer means either dropped logs or a stalled build/request .

Partitioning Strategy

Partition key: `hash(source_id), function_id, or edge_node_id` that generated the line.

Why not project_id

One enterprise project can run thousands of concurrent functions. This kind of partitioning creates the hotspot we are avoiding.

Ordering guarantee

Key by `source_id` preserves per source log ordering. Kafka guarantees order only within a partition

Why source_id works

Each build or function is a bounded or moderate-volume stream. It helps spreads load evenly regardless of tenant size.

Consumer scaling

Consumer groups map one consumer to one partition.

Stream Processing

01 Live Tail

JOB

Read new lines as they arrive and push to active WebSocket viewers for that source_id via Pub/Sub fan-out.

LATENCY TARGET

< 1s. No aggregation, no storage write on this path.

SCALING

Stateless consumers, one per partition-range. Allows horizontal scaling with source count.

02 Windowed Aggregation

JOB

Every 60 seconds, the system calculates error count, log volume, and response times for each project, then writes both raw logs and the summary to storage.

LATE DATA

If a log line arrives a bit late, the system still accepts it for up to 30 extra seconds before finalizing.

SCALING

All logs from the same source always go to the same worker, so that worker's count never gets split or mixed up with another worker's count.

Why Choose Columnar Storage?

DIMENSION	COLUMNAR	ELASTICSEARCH
Data shape	Log data is mostly structured and repeats a lot. That's exactly the kind of data columnar storage shrinks down well.	Elasticsearch saves each log three separate times (for search, for storage, and for fast lookups), so it can't compress this kind of data very well.
Workload	Columnar databases are built to do fast filtering, grouping and lookup.	Elasticsearch is built to search text and rank results by relevance.
Cost at scale	Columnar storage typically shrinks log data by 10-20x, which directly cuts our storage cost from a projected ~140 GB/day.	Elasticsearch needs extra memory and extra index space, so storing the same amount of data for the same amount of time costs more servers.

Elasticsearch is still better for advanced text search. We're okay with that trade-off, because what we actually need is filtering by time range.

Hot, Warm, and Cold Storage

Hot

0–24 hours

Local SSD, uncompacted

Live tail, active debugging

Warm

1–30 days

Compacted, compressed, same cluster

Recent trend queries

Cold

30 days – 12 months

Object storage (S3-class)

Rare, latency-tolerant queries

REASON

After 24 hours, a background job automatically combines many small log files into fewer, larger, compressed files. This is why the warm tier takes up less space and is faster to search than the hot tier.

The system just updates a pointer when data moves between the tiers. When queried, the system automatically checks how old the data is and fetches it from the right place, without the user needing to know the tier.

Batch Export (Vercel Drains Pattern)

-
- | | | |
|-----------|---------------------------------|---|
| 01 | Cold-tier compaction job | A background job takes the warm-tier data and rewrites it into large files and stores them cheaply. These files can be searched directly, without needing to load them into a database first. |
| 02 | Export / drain worker | A separate process reads log data (either from cold storage, or straight from the live stream) and sends copies to outside tools like Datadog or Honeycomb, or to a company's own data warehouse. |
| 03 | Sampling rules | If a user generates a huge number of logs, they can choose to export only a sample of them , instead of everything. |
| 04 | Data warehouse sync | Instead of sending every single log line, send summarized totals (like "500 errors this hour") to tools like BigQuery or Snowflake. |
-

Serve Path

STORAGE —————> CONSUMERS

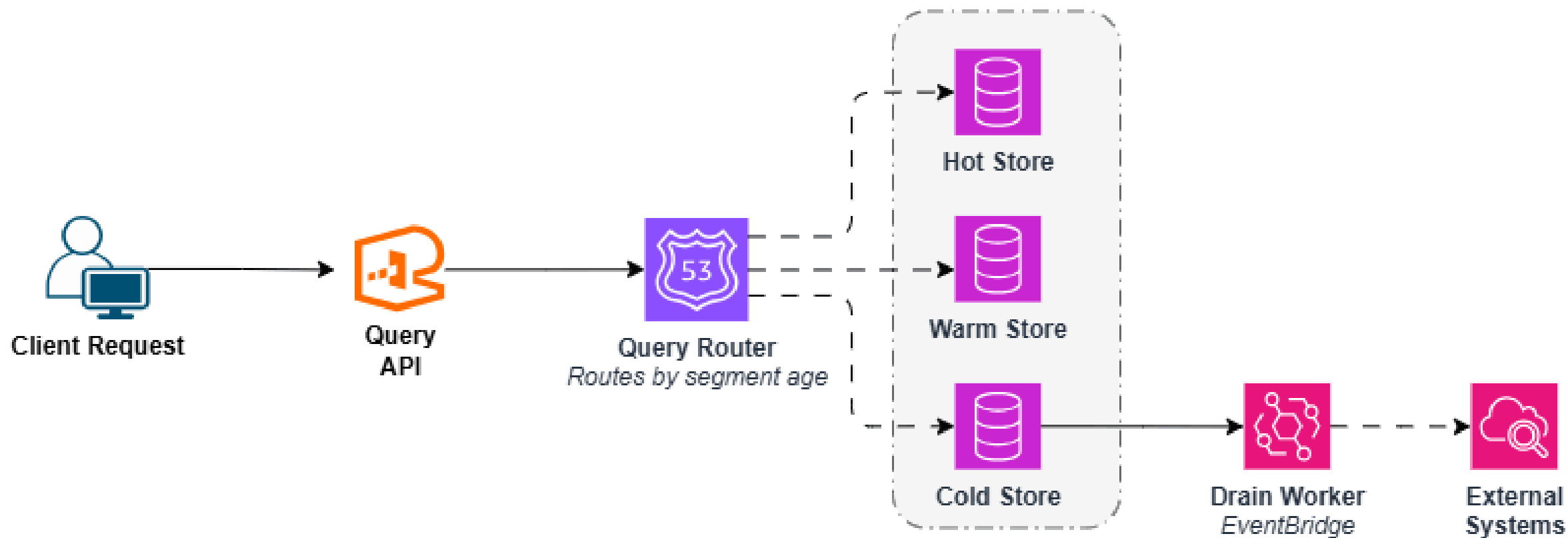
Export

Query

How Logs Get Read Back

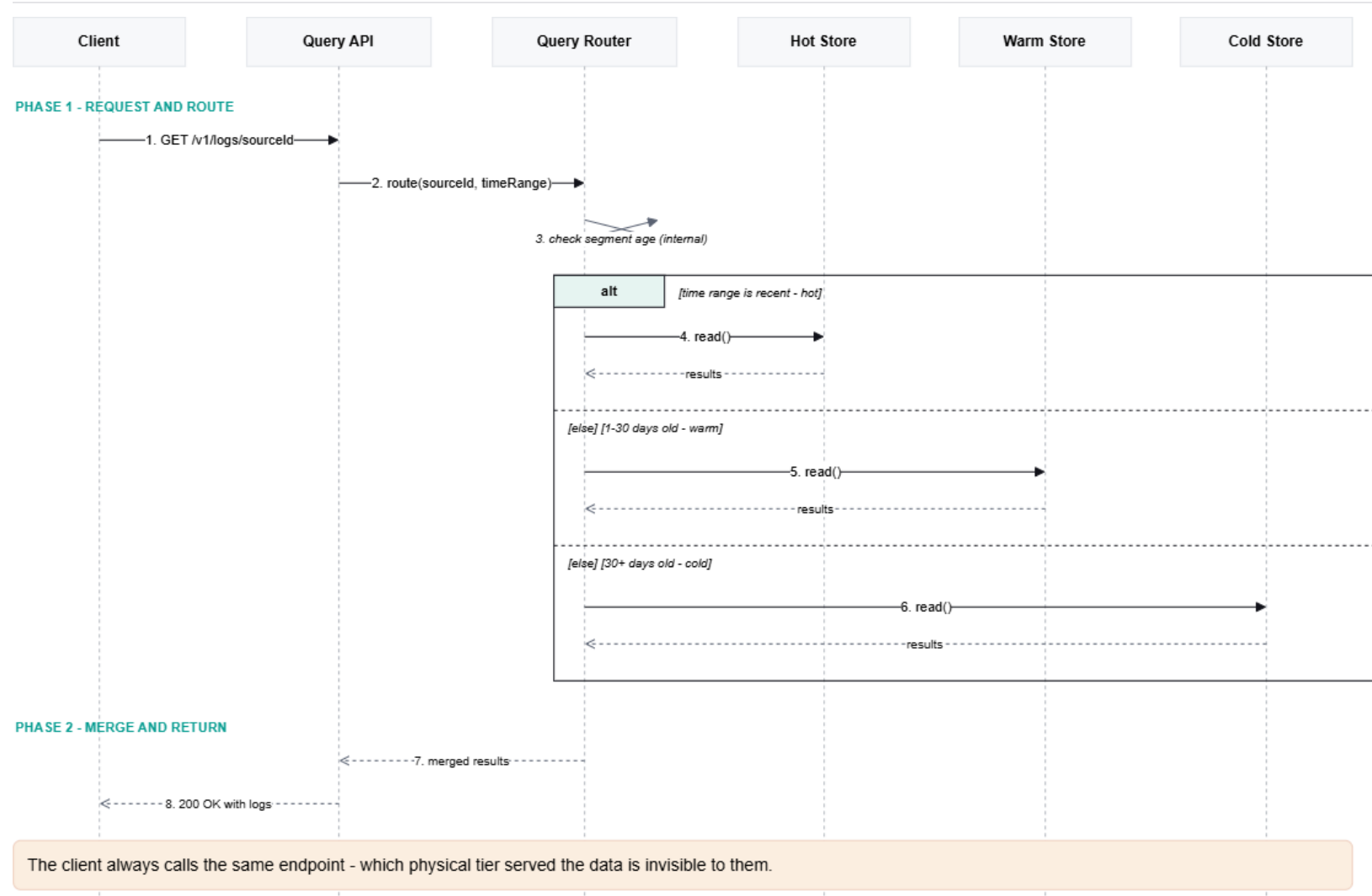
PATTERN	ACCESS SHAPE	SERVED FROM	LATENCY
Live tail	Subscribe by source_id	WebSocket + Pub/Sub fan-out	< 1s
Recent debug lookup	Filter by source_id + time	Hot tier, uncompactd SSD	Sub-second
Trend / aggregate	Group by project, bucketed	Warm tier, columnar segments	1–2s
Historical / compliance	Time range over months	Cold tier, Parquet on S3-class	Seconds

Serve Path Diagram



SEQUENCE DIAGRAM

Historical Query Sequence



Five Years Storage Projection

Grown at ~15% YoY DAU growth and replication factor 3×.

	DAU	COMPRESSED / DAY	LOGICAL (1YR RETAINED)	PHYSICAL (3× REPL.)
Year 1	10.0M	140 GB	51.1 TB	153.3 TB
Year 2	11.5M	161 GB	58.8 TB	176.4 TB
Year 3	13.2M	185 GB	67.6 TB	202.8 TB
Year 4	15.2M	213 GB	77.7 TB	233.1 TB
Year 5	17.5M	245 GB	89.4 TB	268.2 TB

Retention model followed: hot (24h) + warm (30d) stay on cluster storage, cold tier (30d–12mo) moves to object storage at roughly 1/8th the cost per GB.

CONCLUSION

How Log Stream Stays Durable, Fast, and Cheap.

— Buffer, never block

Local buffering + a durable ingestion log means producers never stall waiting on storage.

— Partition by source, not tenant

Spreads load evenly regardless of any one project's size, this avoids the hotspot.

— Tiering is metadata, not rewrites

Hot→warm→cold transitions move segment pointers, this makes system cheap, fast, and reversible.

— Failure is local, not global

Every failure mode degrades one path gracefully.

— Two consumers, two speeds

Live tail optimizes for latency and aggregation optimizes for correctness.

— Columnar over row-store

10–20× compression on log-shaped data turns a 1.4 TB/day problem into a ~140 GB/day one.

— Drains decouple export

External destinations can fail or lag without ever affecting internal durability or query performance.