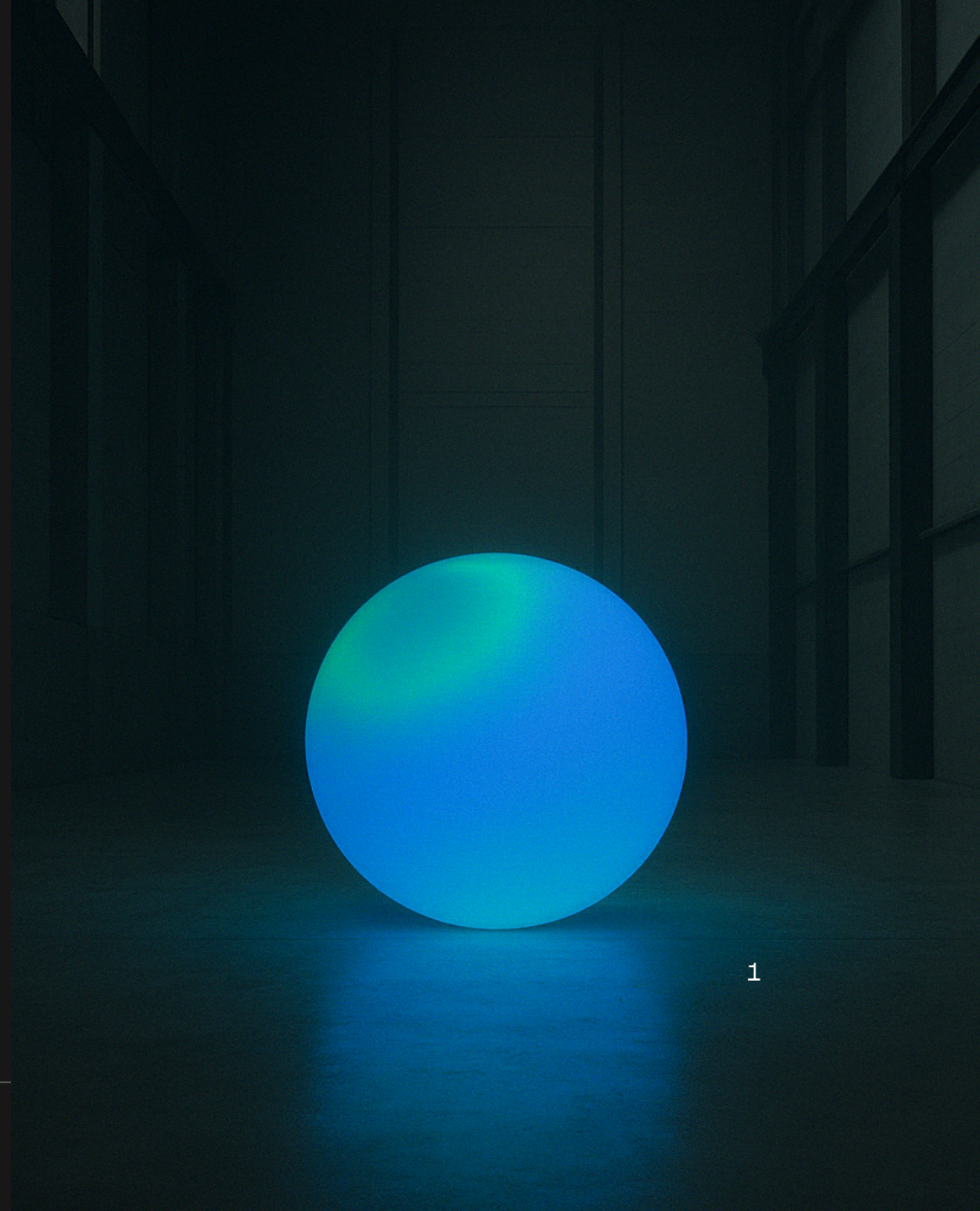
Dune

DuneCP

An event-driven control plane for a blockchain data platform.

Miguel Mascarenhas Filipe, Dune



DEBS 2026

Dune

WHO I AM

I build distributed systems

- Principal Engineer at Dune. I lead the Data Foundation team.
- Amazon DynamoDB launch team.
- Worked on its data-plane, provisioned-throughput and admission control
- Worked on its control plane. Partition management, replica balancing, hotspot mitigation, auto-healing.
- Working on startups since 2015
- Today: control plane for a blockchain data lake.

Dune

WHAT IS DUNE

The source of truth for onchain data

- Public blockchain data is an ocean, and almost unusable raw: fragmented across 100+ chains, hard to decode, always changing.
- Dune does the indexing, decoding, and synthesis. You just write SQL, in DuneSQL, our fork of Trino.

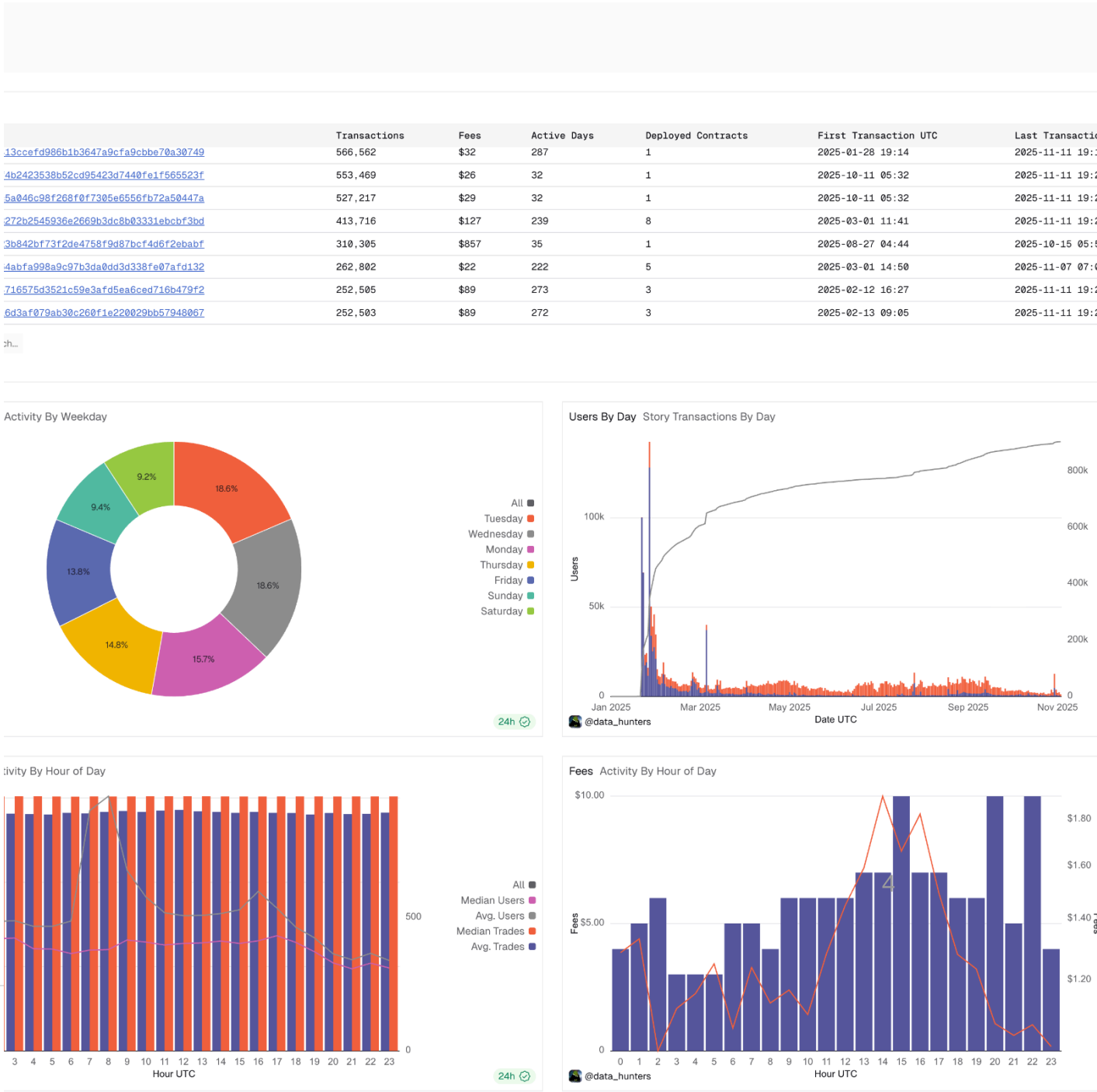
115+ chains. 1.5M+ datasets. 5M+ queries. 180k+ dashboards.

Dune

WHAT PEOPLE BUILD

Query, visualize, share

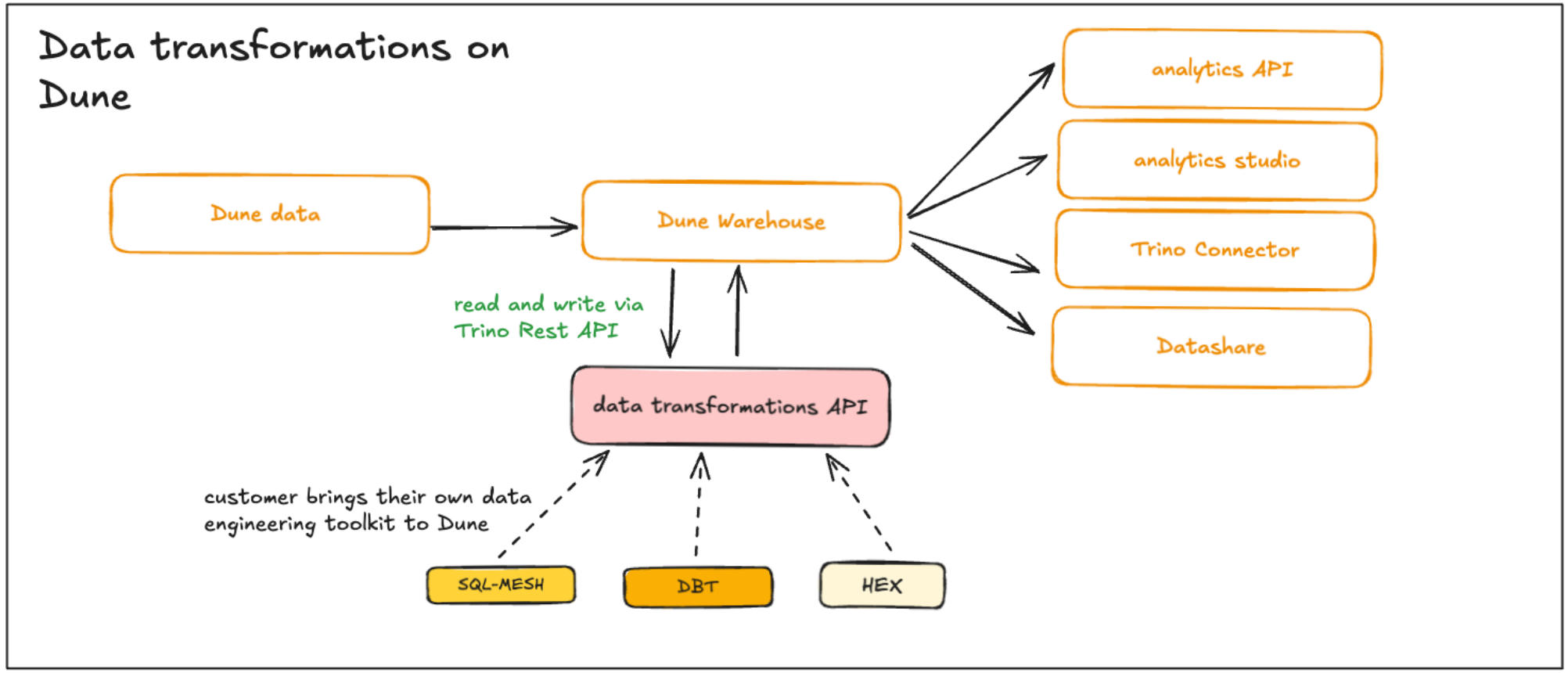
Raw chain history becomes dashboards and APIs, in SQL. No data engineering required.



Dune

WRITE PATH

Bring your own data and tools



Dune

ARCHITECTURE

A data lake, not a database

- Data lives on S3, in the Delta Lake format. Each table is Parquet files plus a transaction log.
- Three engines write the same lake: Spark on Databricks, a Rust ingester, and DuneSQL itself.
- No B-trees. No primary keys. No indexes.
- So how do you keep query performance good on tables that grow without end?

Dune

WORKLOAD

Two kinds of load

- User queries through QES. Some customers run thousands a day to rebuild their models.
- Internal transformation runs far more. Spellbook and dbt rebuild curated data every hour.

~2M BILLABLE EXECUTIONS PER MONTH

- 65K user queries per day.

Dune

THE PRODUCT

One half is governed: the product

- Everything customers touch. SQL endpoints, the API, the app, dashboards, agents.
- Data out: datashares into Snowflake and BigQuery.
- It carries contracts, latency targets, and revenue.
- So it is owned, staffed, and on call.

The other half is not

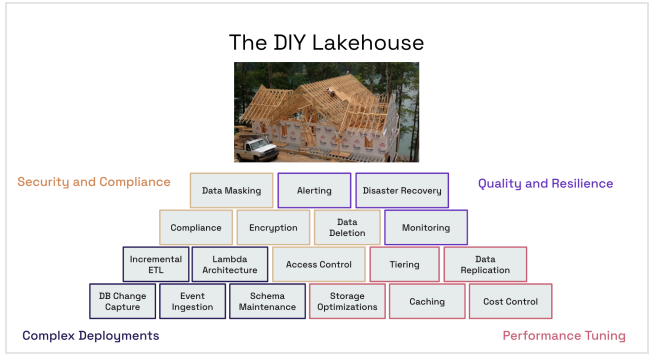
- A second body of work keeps the lake healthy and the books correct.
- Compaction, vacuum, metering, search reindexing, freshness checks, storage accounting, datashare sync.
- Weak SLAs. No product manager. No obvious owner (in a startup).
- And we're still on call..

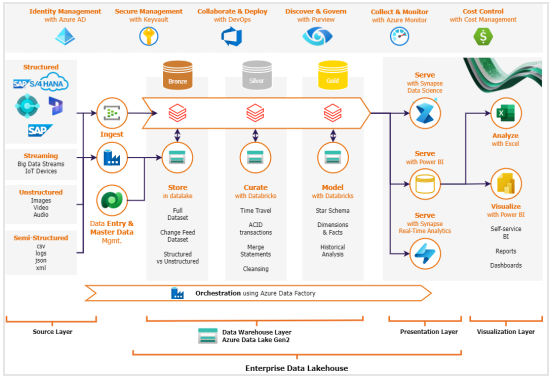
Dune

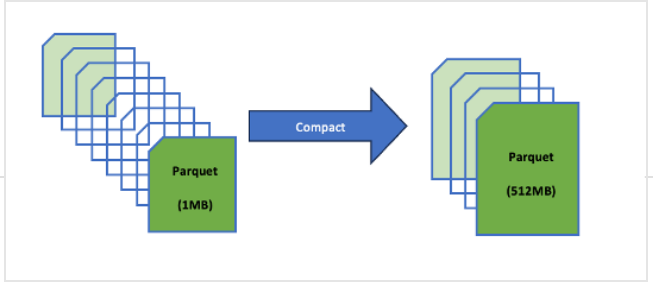
WHY IT EXISTS

A lakehouse is building blocks, not a database

A database does its own housekeeping, invisibly. A lakehouse leaves all of it to you, to assemble and to operate.







Dune

One job, six times over

"Keep Delta tables healthy":

- a Go cron job
- two Rust cron jobs
- Spark on Databricks
- an event-driven Go service
- a SQLMesh deploy hook

Six implementations.
Four runtimes.
Three languages.

All slightly different.

All slightly similar

More than fifty operational jobs in total.

Dune

THE ASK

This has to become one system

- The problem is not only the technical failures. No one sees this as one system.
- A control plane with a holistic view of the work.
- One place engineers look. One practice: encode your operational task as an event.
- Support for the classic problems: retries, liveness, coordination, recovery.

Dune

What it has to handle

Why this is an event-driven problem

- Partition optimizes, incremental table syncs, storage metering, purges: these are naturally event-driven flows.
- Long async activities can be triggered on demand or on schedule
- A data lake is distributed by construction. The metadata is spread across several systems, and many engines touch the same tables.

Jobs to be done, concretely

- Datashare: incrementally replicate tables into customer systems (Snowflake, BigQuery).
- Layout at rest: tables are Parquet files on S3. Good performance needs file statistics, recompaction, repartitioning, reclustering. External merge-sorts over a whole table.
- Table logs: the Delta logs need health checks and maintenance.
- Metadata churn: about one million table creations and tombstones every day.
- Purge: deleted datasets have to be reclaimed from S3.

Textbook Problems

- It is easy to reach for events and queues and forget flow and queueing theory.
- Two failures recur in every system like this: the poison pill, and throughput collapse.
- We have both. When they hit, maintenance is hampered and we get slow burning fires that progressively degrade quality of service.

Dune

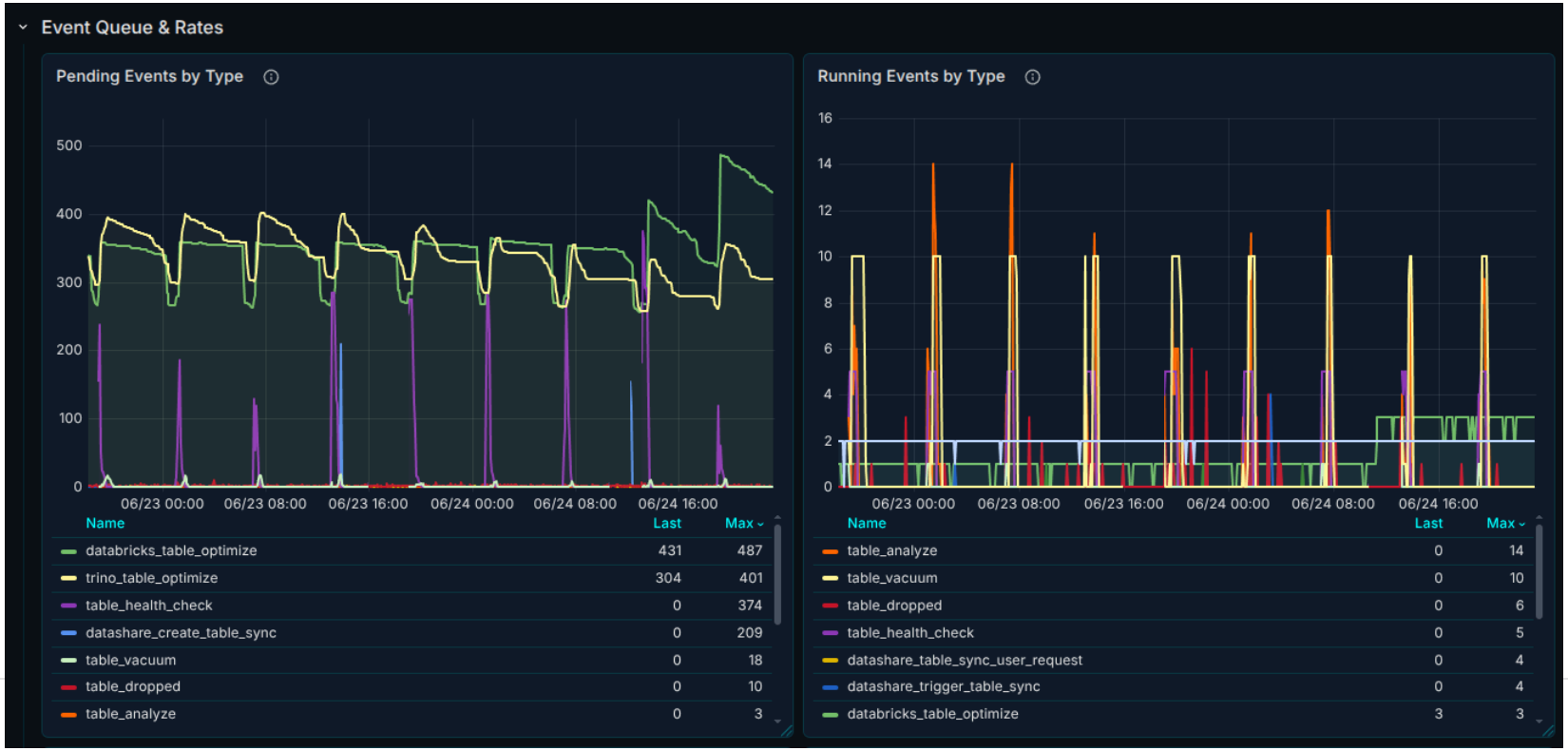
POISON PILL

The poison pill

- A health check flags table A for OPTIMIZE. Building the plan scans the table's partitions, and on a degenerate table that scan wedges the shared query engine.
- Table A is maybe one in a ten thousand. It still took down the whole optimize loop with it.
- Even a health check can kill the worker running it. One bad table poisons everything that touches it.
- Real, recurring incidents in our tracker: DF-520, DF-486, DWH-3.

The infinite backlog

- Work arrives faster than the backend can drain it. The queue never empties, and for days fast-growing tables go unoptimized while queries degrade.



Dune

A tiny control plane

What do we need

- One control plane for the background workflows that keep DuneSQL healthy: durable state, retries, coordination, event intake, observability.
- Centralize maintenance for tables, materialized views, and views.
- Heal tables before performance degrades.
- Emit events on table state changes; expose APIs for metering and billing.
- The boundary is the discipline. This is not a generic scheduler for all of Dune: user-facing schedulers, infra janitors, and heavyweight ingestion stay outside.

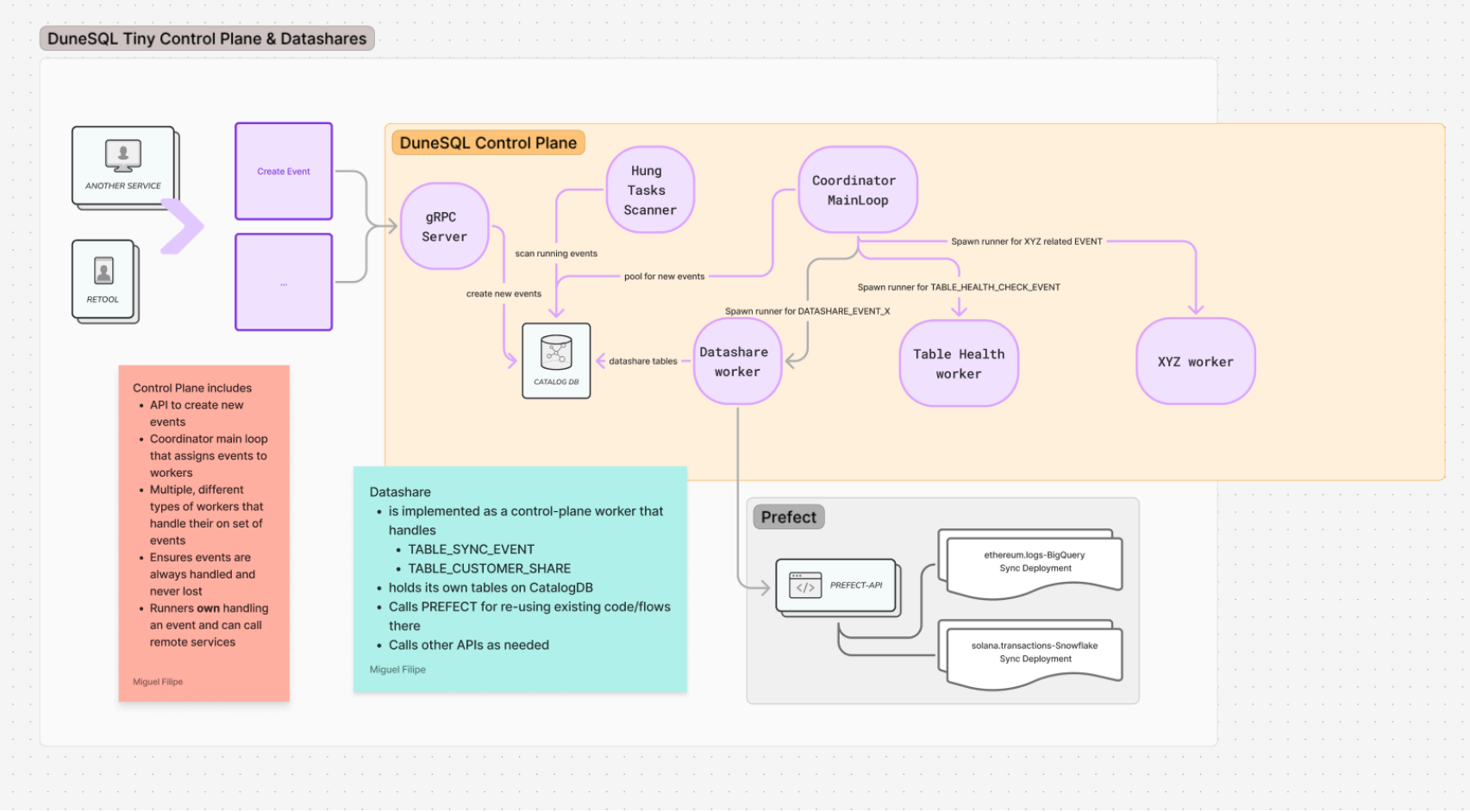
Building blocks

- Four things: the Event, the Lease, a single Coordinator, a small state machine.
- Target load: ~10k events per hour
- State lives in Postgres. The event queue is a Postgres table. That is the whole substrate.
- Start simple: one centralized, singleton coordinator.
- Migration is trivial. An existing cron job just creates an Event.

Dune

ARCHITECTURE

The shape of the system



The Event

```
type Event struct {
    ID              string              // ULID
    Type            string              // "table_health_check", ...
    State           EventState          // PENDING -> RUNNING -> COMPLETED / FAILED
    Payload         []byte              // opaque, worker-defined
    OutputMetadata map[string]string
    WorkerID        *string             // lease owner
    ExpiresAt       time.Time
    LastHeartbeat   *time.Time          // liveness
}
```


- Four states. PENDING → RUNNING → COMPLETED or FAILED.

Properties

- Events are independent. No coupled multi-event transactions.
- Anyone can create an event over gRPC.
- Processing one event can create more events.
- One worker owns an event at a time.
- Workers prove liveness with heartbeats.

Workers

- In-process: the coordinator runs a goroutine per known event type.
- Fault-tolerant: a worker owns the lease and enforces it on every state change and heartbeat.
- Distributed: for engine or runtime-specific work that cannot live in the Go binary. Designed for from day one.

Fault tolerance

- Leases and heartbeats: a worker holds a lease and proves liveness
- Conditional updates, gated by the lease token: every state change asserts the caller still owns the event.
- Idempotent steps, bounded retries, at-least-once: handlers must be safe to repeat.
- Exponential backoff and expirations: failure is bounded, not infinite.

External workers

- Resource: OPTIMIZE is embarrassingly parallel and compute-intensive. It should not share the coordinator's pod.
- Polyglot: candidate workers are Java/Spark, Rust. The contract has to be language-agnostic.
- Blast radius: a misbehaving worker must not destabilize the coordinator.
- Deployment: a new worker version should not force a coordinator redeploy.

Dune

The API contract is five operations

- `CreateEvent`
- `ClaimEvent`
- `HeartbeatEvent`
- `CompleteEvent`
- `FailEvent`

Dune

WHAT WE DID NOT NEED

"All you need is Postgres" :-)

- No Kafka.
- No Temporal. No Flink.
- No distributed transactions.
- No new cluster to operate.
- The difficulty was in reducing scope: finding the minimum cut.

Dune

In production

Dune

DATASHARE

A second domain: datashare

- Customers want Dune datasets replicated into their own Snowflake or BigQuery.
- DuneCP does not move the data. Prefect runs the replication; DuneCP owns the lifecycle: provision the vendor share, trigger the run, poll for completion, handle failure.
- One target abstraction hides the vendors: Snowflake shares, BigQuery Analytics Hub, the same workflow.
- This is the proof the model generalizes. The queue that heals tables also ships datashares.

Dune

DATASHARE

One request becomes a chain of events

- A user request decomposes into a chain of independent child events:
  - `user_request` → `create_table_sync` → `trigger_table_sync` → `create_share`
- Each child is independently retryable, observable, and debuggable.
- The parent polls for completion and keeps its own heartbeat while it waits.
- Billing rides a final `datashare_table_sync_completion` event: bytes and CPU time, with an idempotency key.

Dune

MAINTENANCE

Maintenance, routed to the engine that wrote the table

- Most blockchain tables are written by Databricks, not Trino.
- You cannot safely OPTIMIZE a Databricks-written table from Trino: concurrent writers corrupt it.
- So the event picks the engine. Trino-written tables optimize in-process; Databricks-written tables emit `databricks_table_optimize`, dispatched to a Databricks job.
- Same event model, same fault tolerance. The external-worker contract from the last act, in production.

Dune

ONE HOME

The live control plane

One dashboard for every loop.

Dune

THE CONSTRAINT

Built and run by two people

- Two engineers, later three.
- More than fifty operational concerns now share one home, one set of primitives, one place to look and improve.
- No new system to operate: Postgres, plus the runtimes we already ran.
- We want to drop what we have on Prefect.
- Why isn't there a simple Open Source solution for this? What am I missing?

Dune

What is still hard

Dune

VISIBILITY

Centralizing makes the problems obvious

- Throughput collapse and backlog were always there. Spread across runtimes, hard to see them.
- One queue, one shared chart: the moment work arrives faster than the backend drains, the backlog line climbs and stays there.
- That is the first thing the control plane bought us.
  - Not a fix: a place where the problem is undeniable.
- Secondly: shared understanding of classic challenges

Dune

NO GENERIC CURE

But the fix is case-by-case, per event type

- There is no single knob. Each event type degrades differently and gets its own remedy.
- Optimize and health: per-table timeouts, bounded concurrency, a dead-letter for poison tables.
- Load: shed work by skipping inactive tables instead of scanning the whole lake.
- Anti-fragility mechanisms:
  - Shuffle scan order, shuffle batches.
  - Dynamic batch ajustments with heuristics
  - Specific defensive measures against known degenerate case.

Dune

SCALE SURPRISES

The delete-table flood

- Dropping datasets creates `table_dropped` events. At peak, hundreds of thousands a day.
- System designed for ~1k-10k events per day, not ~500k-5M a day.
- One event per dropped table stopped scaling: the queue filled with tombstone work.
- The fix was a batch event type: claim and process drops in bounded batches, not one at a time.
- The scale of a single workload reshaped the model. You do not see this until it is all in one place.

One coordinator, one blast radius

- The coordinator is a singleton. Simple to reason about, simple to operate.
- The cost: its blast radius is the whole control plane. Today we lean on restart-on-failure.
- Real high availability for the coordinator is still ahead of us.
- So are the planned directions: more external workers, and a bridge so every cron job becomes an event.

Why we built it instead of adopting one

- Adopting an existing engine would have been more work, not less: a bigger impedance mismatch to settle than the few primitives we actually needed.
- The hard part is modeling the control-plane workflows correctly. That work is unavoidable, whatever tool you pick.
- Had we built these workflows on Airflow or Prefect, we would still have to audit every sub-step for idempotency, at-least-once semantics, restart, and liveness.

Dune

THE TAKEAWAY

Minimal, but sufficient

- DuneCP is stupid simple. That simplicity forces one shared design language for resilient workflows.
- Its few features make engineers confront the classic problems and patterns up-front, not after an incident.
- It does not need much. The semantics it has are easy, transparent, and sufficient.

Dune

Questions?

miguel@dune.com

THANK YOU TO THE DEBS 2026 ORGANIZERS