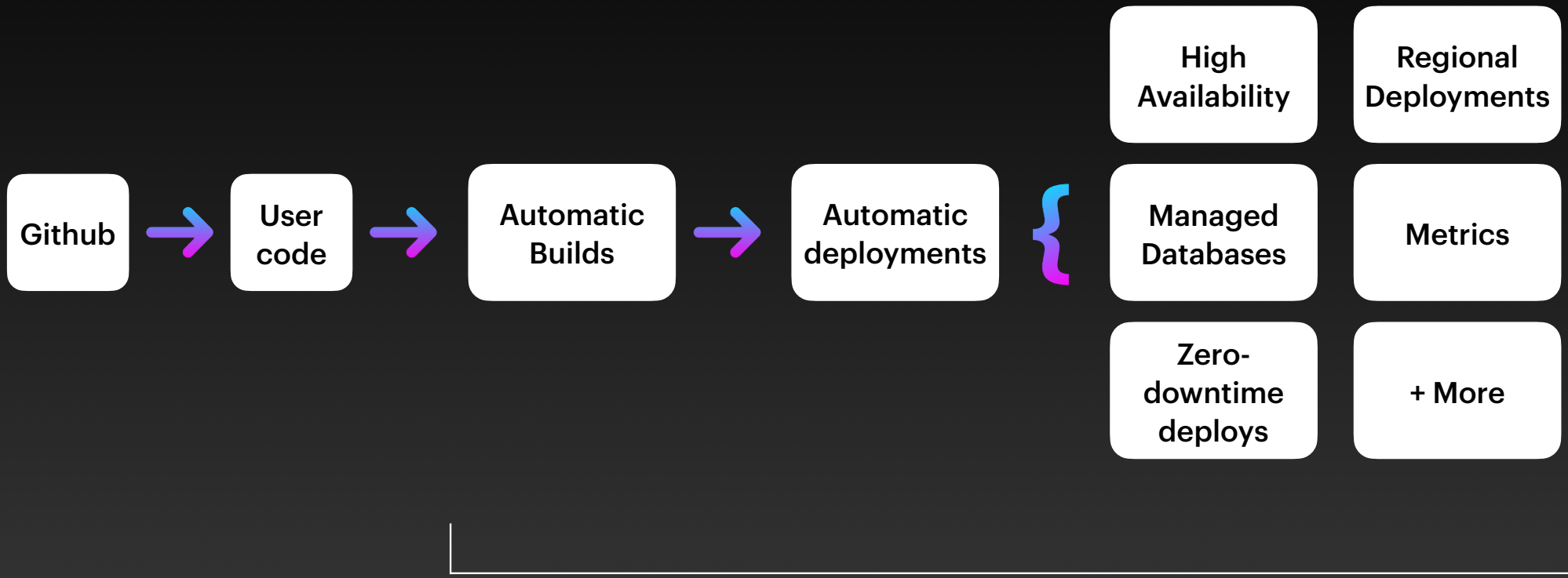
ClickHouse

Supercharging billing-efficiency With ClickHouse

Tom Watson - 2023

First - who we are, and what we do

What FLO does:



Architecture

Heavy users of Kubernetes (K8s)

Our infrastructure is split into 2 main partitions:

- “Services” cluster - internal applications, build-functionality, core web-app, user-service-allocation, etc
- “Client” cluster - client workloads, monitoring, client workload operator. One of these per-region.

Client isolation is provided by extensive use of K8s Namespaces, Cilium Network policies and other built-in isolation tools.

Challenges + Requirements

Variety of data

- Client workloads can use a range of CPU + Memory, varying by service.
- There can be a *lot* of workloads on a cluster.

Challenges + Requirements

Complexity

Pods can:

- crash or reboot
- be rescheduled across nodes
- be re-deployed
- be scaled up or down
- Certain identifiers can change on reboot

and this can happen at any moment!

Challenges + Requirements

Correctness + Reliability

- We can’t overcharge or mis-charge customers.
- In the event of disputes or other issues (or even just sanity checking), we need to have a record of all the data used to derive a charge.
- Business + billing rules can change more rapidly than even the highest velocity stream, so this logic needs to be decoupled from the mechanics underneath, and cleanly verifiable.
- Minimise possible runtime issues.
- Outages, crashes and interruptions are a fact of life, strong fault-tolerance and “self-healing” semantics are a must.

Challenges + Requirements

Throughput + Velocity

- Latency is important, up-to-date bills are a desirable + useful feature
- Data volumes can be large and frequent
  - We need multiple metrics, per-sample timestamp, per pod, many of which need to be joined with other streams, and aggregated up to the levels of business rules.

Challenges + Requirements

Putting these all together, what features do we need from our system?

Challenges + Requirements

Immutable, log structured data store

Latency sensitive/Read-Write Performance

Reliable + ops-friendly

Fault tolerance + resilient

Ease of adjusting business-logic

Challenges + Requirements

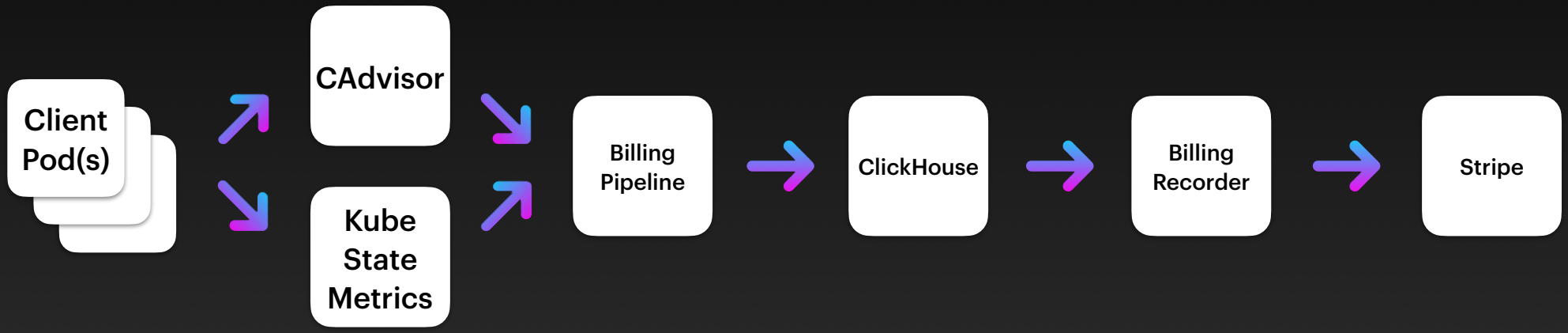
Data Immutability + log-structured tables:

- ClickHouse's LSMT tables are a perfect fit here
- Column oriented data store is again, the ideal solution here
- Read + Write performance:
  - We need to be able to sink 10k samples/second minimum

Challenges + Requirements

- Need to expect faults up+downstream
- Gracefully degrade, or tolerate downtime without blowing up
- Minimise moving parts + edge cases to reduce potential for corrupt data
- Writes should be idempotent, deduplicated and safe to retry
- Metrics + alerts

Data-Flow


Client Pod(s)
CAdvisor
Kube State Metrics
Billing Pipeline
ClickHouse
Billing Recorder
Stripe

Pipeline Details

CAdvisor

Provides a *huge* number of container-level operational metrics

Network in/egress, cpu, memory, storage IO, everything

Kube State Metrics

“metadata” metrics, that tie containers up to

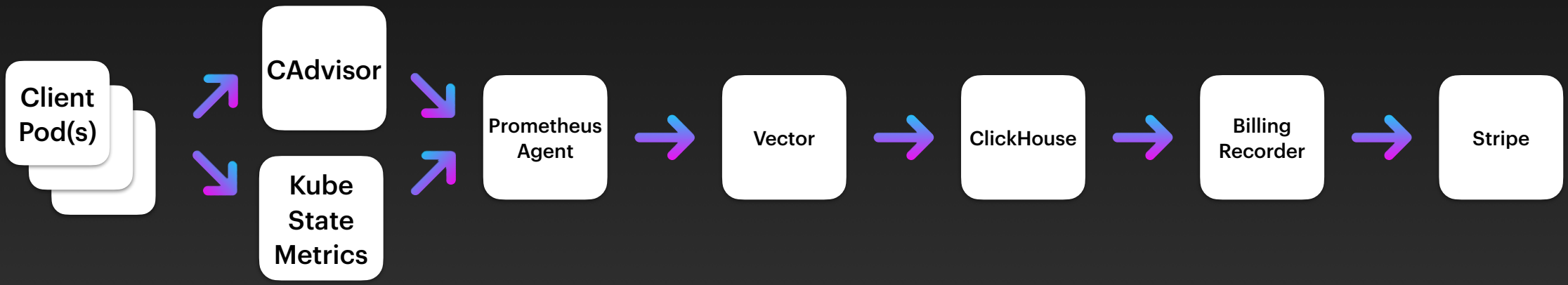
Their parent namespace + workspace in the FLO “hierarchy”

Aside: Prometheus Metrics

```
# TYPE fl0_version counter
fl0_version{service="billing_ingestor",version="2.0.2504+sha.3f3a7bc"} 1

# TYPE billing_ingestor_remote_write_success counter
billing_ingestor_remote_write_success 246856

# TYPE billing_ingestor_batcher_metric_total counter
billing_ingestor_batcher_metric_total{metric_type="network_egress"} 21934993
billing_ingestor_batcher_metric_total{metric_type="other"} 20887140
billing_ingestor_batcher_metric_total{metric_type="pod_id"} 9530400
billing_ingestor_batcher_metric_total{metric_type="pod_cpu_limit"} 16170888

# TYPE billing_core_insert_pod_id_success counter
billing_core_insert_pod_id_success 374

# TYPE billing_core_insert_container_network_egress_success counter
billing_core_insert_container_network_egress_success 1679

# TYPE billing_core_insert_pod_id_total counter
billing_core_insert_pod_id_total 374

# TYPE billing_core_insert_container_network_egress_total counter
billing_core_insert_container_network_egress_total 1680

# TYPE billing_ingestor_remote_write_total counter
billing_ingestor_remote_write_total 246856

# TYPE billing_ingestor_healthcheck_total counter
billing_ingestor_healthcheck_total 74053

# TYPE billing_core_insert_pod_cpu_limit_total counter
billing_core_insert_pod_cpu_limit_total 804

# TYPE billing_ingestor_metric_batch_write_total counter
billing_ingestor_metric_batch_write_total 2858

# TYPE billing_core_insert_pod_cpu_limit_success counter
billing_core_insert_pod_cpu_limit_success 804

# TYPE billing_ingestor_metric_batch_sync_data_total counter
billing_ingestor_metric_batch_sync_data_total 740568

# TYPE billing_core_insert_container_network_egress_duration_seconds histogram
billing_core_insert_container_network_egress_duration_seconds_bucket{le="0.005"} 0
```

- Prometheus Metrics are a text based format for metric data.
- Widely used, straightforward to parse
- Applications export them - usually by exposing a `/metrics` endpoint
- Low overhead

Version 1 Pipeline



Version 1 Pipeline In Depth

Prometheus Agent

- slimmed down version of Prometheus
- Scraped metrics and forwarded them on via it's "Remote Write" protocol
- If the remote-write fails to acknowledge, it buffers them onto disk

Vector

- Fantastic tool for collecting + routing + filtering logs + metrics
- We use it for controlling the number + cardinality of the metrics we persist
- For billing, we have it write filter out metrics we don't care about, and write them to a ClickHouse Table in Newline Delimited JSON format

Version 1 Pipeline In Depth

ClickHouse

Metrics are written in batches into a ClickHouse table:

We make heavy use of ClickHouse's *extremely* useful Materialised Views implementation to parse the information we want into sub-tables.

Billing Recorder

Custom service we wrote in Rust - grabs valid subscriptions from Stripe, runs the "business logic SQL queries" against ClickHouse and records the results in Stripe. This is the only place where our "business logic" lives.

Data Samples


```
{
    "name": "kube_pod_labels",
    "tags": {
        "instance": "10.16.30.105:8080",
        "job": "kube-state-metrics",
        "label_fl0_dev_env": "prod",
        "label_fl0_dev_organization_id": "d7a8364e-cbb7-4e3a-8df6-0cf012d59a1b",
        "label_fl0_dev_project_id": "happy-antelope-b537473b",
        "label_fl0_dev_resource_id": "poc-api",
        "namespace": "c-happy-antelope-b537473b-prod",
        "pod": "poc-api-6b6f9d4f9f-p52rj",
        "uid": "25e6237b-18c9-4f46-8674-593853004cfc"
    },
    "timestamp": "2023-07-16T23:03:57.978Z",
    "kind": "absolute",
    "gauge": {
        "value": 1.0
    }
}
```

Data Samples


```
{
    "name": "container_network_transmit_bytes_total",
    "tags": {
        "id": "/kubepods.slice/kubepods-burstable.slice/kubepods-burstable-p...",
        "image": "registry.k8s.io/pause@sha256:7031c1b283388d2c2e09b57badb803c05ebed362dc88d84b480cc47f72a21097",
        "instance": "i-0e6fb613bcf654d67",
        "interface": "eth0",
        "job": "cadvisor",
        "name": "7fa53a938a223f64c74c6ce5090d30ab003f047556a36f2b38fef8f8c580d0e3",
        "namespace": "c-iron-potato-09c1f9cc-dev",
        "pod": "fl0testing-7766f657bf-r4s8m"
    },
    "timestamp": "2023-10-12T22:03:27.715Z",
    "kind": "absolute",
    "gauge": {
        "value": 1016.0
    }
}
```

Hang on a moment...

Performing Magic with Inline Parsing

```
CREATE TABLE metrics
(
    metric    String CODEC (ZSTD),
    name LowCardinality(String)      DEFAULT visitParamExtractString(metric, 'name'),
    timestamp DateTime64             DEFAULT parseDateTime64BestEffort(visitParamExtractString(metric, 'timestamp')) CODEC (DoubleDelta),
    kind LowCardinality(String)      DEFAULT visitParamExtractString(metric, 'kind'),
    gauge     Float64                DEFAULT visitParamExtractFloat(visitParamExtractRaw(metric, 'gauge'),
                                                                     'value') CODEC (Gorilla),
    job LowCardinality(String)       DEFAULT visitParamExtractString(visitParamExtractRaw(metric, 'tags'), 'job'),
    namespace LowCardinality(String) DEFAULT visitParamExtractString(visitParamExtractRaw(metric, 'tags'), 'namespace'),
    pod LowCardinality(String)       DEFAULT visitParamExtractString(visitParamExtractRaw(metric, 'tags'), 'pod'),
    project_id                       DEFAULT visitParamExtractString(visitParamExtractRaw(metric, 'tags'),
                                                                     'label_fl0_dev_project_id') CODEC (ZSTD),
    resource_id                      DEFAULT visitParamExtractString(visitParamExtractRaw(metric, 'tags'),
                                                                     'label_fl0_dev_resource_id') CODEC (ZSTD),
    workspace_id                     DEFAULT visitParamExtractString(visitParamExtractRaw(metric, 'tags'),
                                                                     'label_fl0_dev_organization_id') CODEC (ZSTD),
    environment                      DEFAULT visitParamExtractString(visitParamExtractRaw(metric, 'tags'),
                                                                     'label_fl0_dev_env') CODEC (ZSTD)
) ENGINE = MergeTree ORDER BY (name, pod, timestamp);
```

Performing Magic with Inline Parsing

- JSON is easy + ubiquitous
- Offloading this to the DB is a massive developer win
- Being able to define the logic in SQL is a massive win
- Minimises the number of moving parts + external systems

Performing Magic with Incremental Views

```
CREATE TABLE usage_limits
(
    timestamp    DateTime64 CODEC (DoubleDelta),
    pod          String CODEC (ZSTD),
    workspace_id String CODEC (ZSTD),
    limit_type LowCardinality(String),
    gauge        Float64 CODEC (Gorilla)
) ENGINE MergeTree ORDER BY (workspace_id, pod, timestamp);

CREATE MATERIALIZED VIEW usage_mv TO usage_limits
AS
WITH T1 AS (SELECT DISTINCT pod, workspace_id
            FROM workspace
            WHERE pod =!= '')
SELECT timestamp,
       pod,
       T1.workspace_id                                                    AS workspace_id,
       visitParamExtractString(visitParamExtractRaw(metric, 'tags'), 'resource') AS limit_type,
       gauge
FROM metrics
         INNER JOIN T1 ON T1.pod = metrics.pod
WHERE limit_type =!= '';
```

Performing Magic with Incremental Views

```
CREATE TABLE usage_limits
(
    timestamp    DateTime64 CODEC (DoubleDelta),
    pod          String CODEC (ZSTD),
    workspace_id String CODEC (ZSTD),
    limit_type LowCardinality(String),
    gauge        Float64 CODEC (Gorilla)
) ENGINE MergeTree ORDER BY (workspace_id, pod, timestamp);

CREATE MATERIALIZED VIEW usage_mv TO usage_limits
AS
WITH T1 AS (SELECT DISTINCT pod, workspace_id
            FROM workspace
            WHERE pod =!= '')
SELECT timestamp,
       pod,
       T1.workspace_id                                                    AS workspace_id,
       visitParamExtractString(visitParamExtractRaw(metric, 'tags'), 'resource') AS limit_type,
       gauge
FROM metrics
         INNER JOIN T1 ON T1.pod = metrics.pod
WHERE limit_type =!= '';
```

Materialised view but super-charged

MV's auto-update: remove dependence on external systems for refreshing them. i.e. no spark/dbt/ matillion required.

Billing Recorder

- Core logic is independent of how often it’s run
- Reads Stripe for active subscriptions + their start date
- For each billable metric, queries our tables in Clickhouse for the total amount between the start of the active billing-period and now.
- Writes the total into Stripe
  - Writing totals instead of deltas gives us a degree of write-idempotency

Billing Recorder

- If it dies at any point in its execution, it's safe to re-run it
- Concurrent running copies don't invalidate correctness of recorded data, as we effectively take "last write wins".
  - Similar to "convergence" property from CRDTs

Billing Recorder

- Resilient to upstream “stalls” or intermittent DB/network outages as the next time it runs after upstream recovers, will re-converge on writing the correct data to Stripe.
- It emits Prom-metrics itself, which are wired into alerts
  - Helps us a catch a huge number of those “grey-failure” cases that often occur in data pipelines

Billing Recorder

- Written in Rust, so it's fast + efficient
  - Average billing cycle duration is about ~100 seconds in production.
  - Memory usage is < 100mb
- Result<T,E> types + no-exceptions makes writing reliable software a lot simpler.
- Strong type system permits pushing many runtime checks into compile-time checks
  - "If it compiles, it works".
- Client usage is queried and recorded asynchronously
  - Minimises peak memory usage
  - Doesn't compromise on throughput

Pros + Cons

- Satisfactory + functional V1
- DB was being used by more applications
- Dev speed/runtime-performance tradeoff
  - V1 pipeline was incurring more load than we wanted
- The initial table designs suited the billing queries fine, but were a poor fit for the newer workloads we were putting on them
- Write latency was suffering..
  - Back p

Pros + Cons

Write latency was suffering...

- Back pressure was working
- But not how we wanted...
- And the components didn’t give us the required dials to tune it

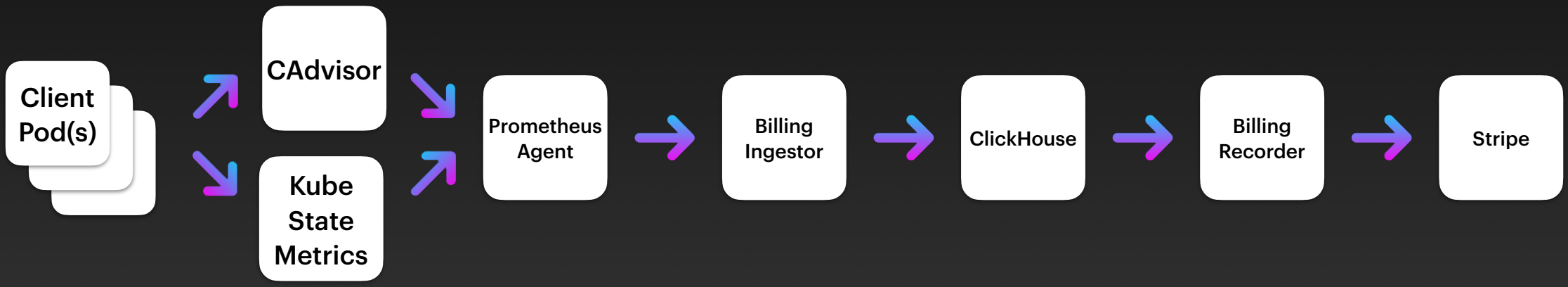
The Solution?

Rewrite-It-In-Rust!

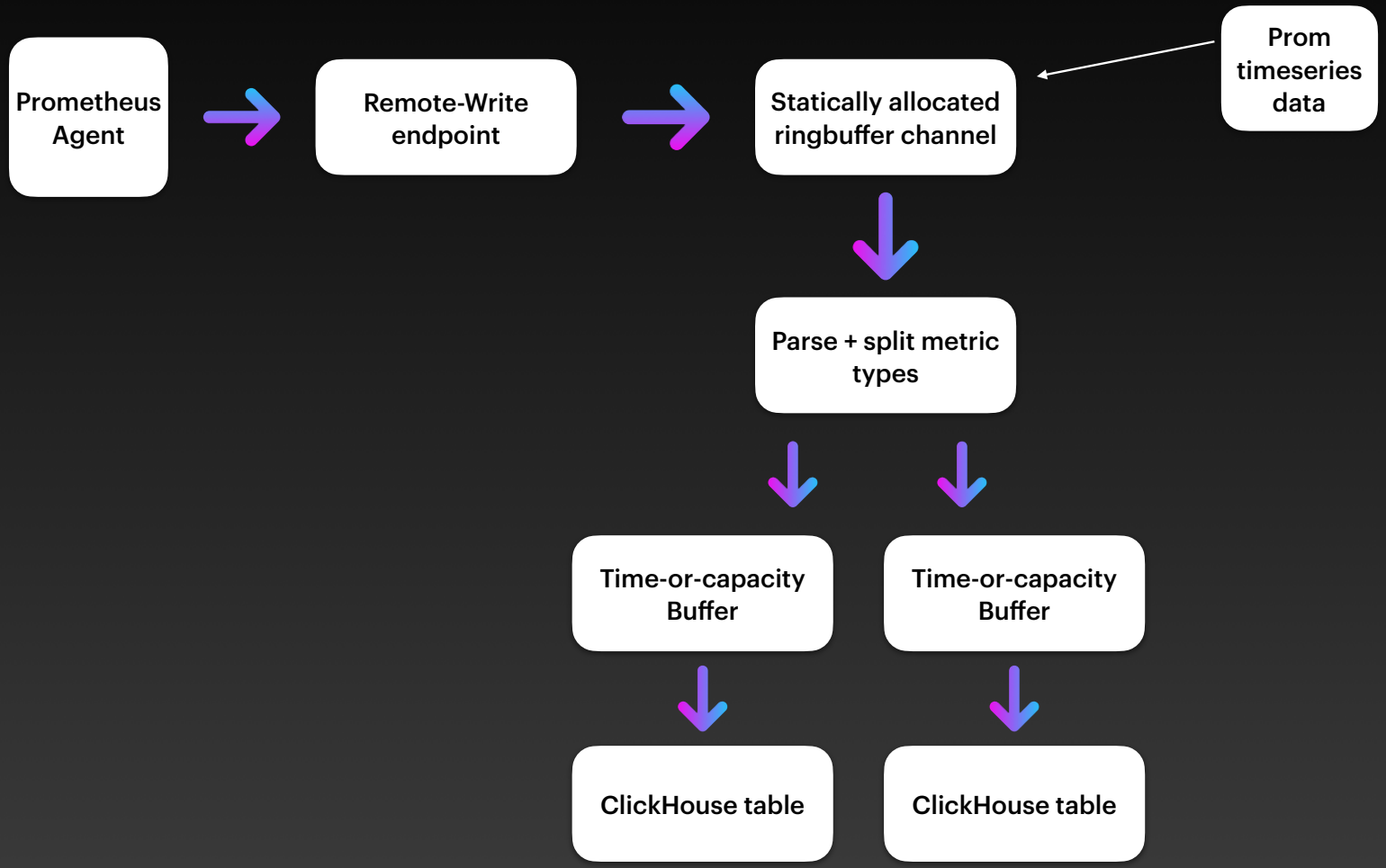
Rewrite-It-In-Rust!

(Only sort of joking)

Version 2 Pipeline



Version 2 Pipeline



Version 2 Pipeline

2 people.

2 days flat.

~1,500 lines of Rust + tests + metrics, dashboards + alerts

Implemented the Prom Remote Write API

Parse raw prometheus events into more compact Rust structs

Filtering + splitting logic all under our control now.

Redesigned the tables to make the DB happier + faster.

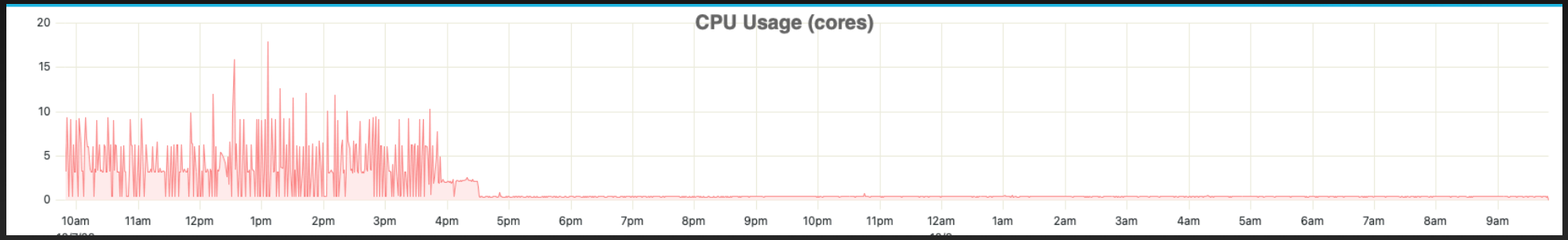
Version 2 Pipeline

```
CREATE TABLE pod_labels
(
    raw_ts        DateTime64,
    timestamp     DateTime64 DEFAULT toStartOfMonth(raw_ts),
    pod           String,
    service_id    String,
    project_id    String,
    workspace_id  String,
    namespace     String,
    env_id LowCardinality(String),
    INDEX ws_idx workspace_id TYPE set(0) GRANULARITY 1,
    INDEX pod_idx pod TYPE set(0) GRANULARITY 1
)
    ENGINE ReplacingMergeTree
        PARTITION BY toYYYYMM(ts_normalised)
        ORDER BY (workspace_id, pod, ts_normalised);
```

Performance Comparison

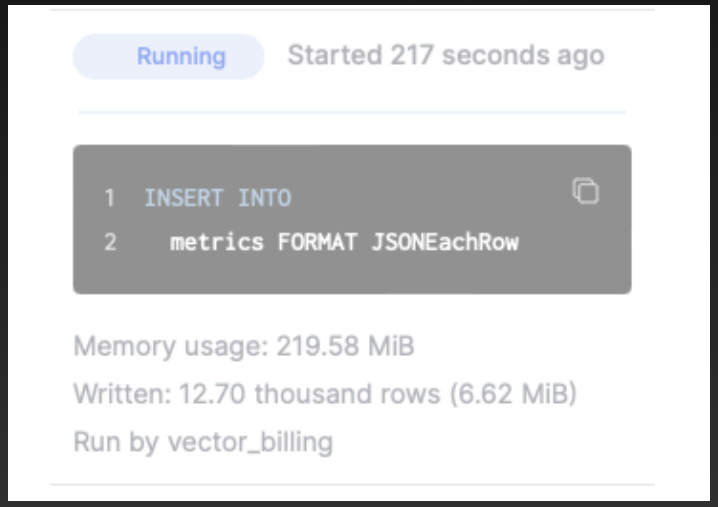
A picture is worth a thousand words...

Performance Comparison



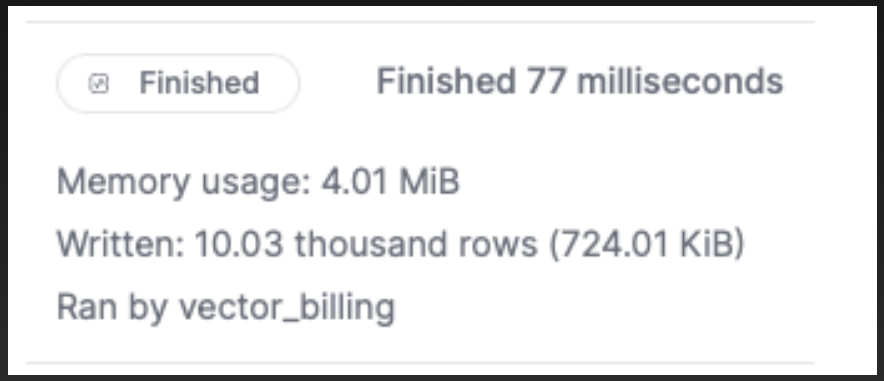
Performance Comparison

V1 pipeline:



Performance Comparison

V2 pipeline:



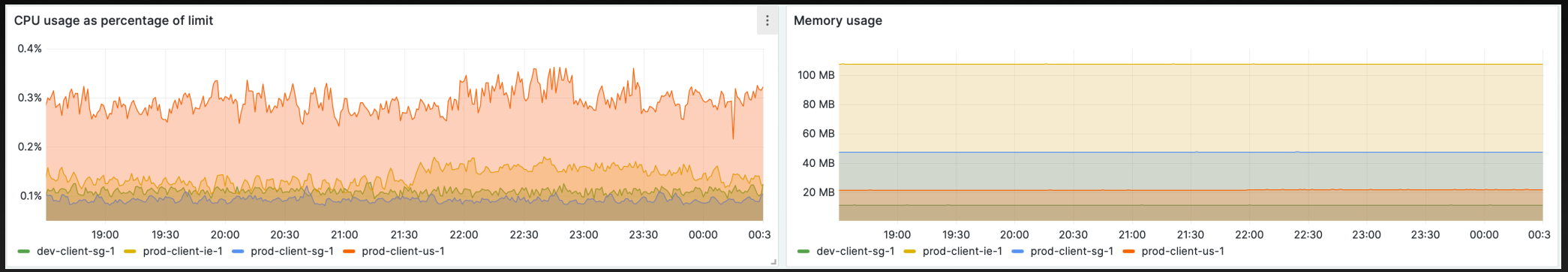
What Changed?

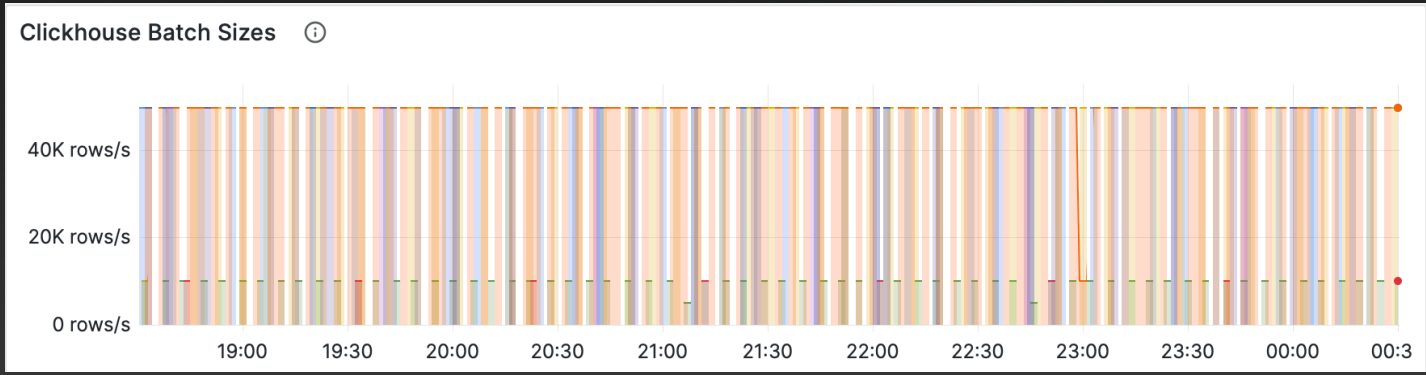
- Better designed tables + writing directly to them.
- The JSON parsing was a killer feature that held us over until we had the bandwidth to swap to a more efficient design upstream.
- The billing queries now need only read a single part on-disk, and most have sub-100ms latency.
- Smarter use of appropriate Table Engines for your workload: ReplacingMergeTree worked wonders for some of our tables.
  - Deduplicating keys off the hot-path reduced query load, *without* impacting write or query performance.

What Changed?

- clickhouse.rs Rust library uses the far more efficient RowBinary format:
  - More db friendly format than JSON - plaintext can have massive overheads
  - Compact binary format fit more rows, in less space, with less db overhead.
- The only limiting factor now, is Prometheus itself not having any more data to hand off.
- Increased control over back-pressure scenarios

Some nice graphs





Summary

- If you need to save dev time, or your workload fits it well, Materialised Views are a super handy feature, more DBs need the auto-updating functionality!
- Make extensive use of the `EXPLAIN` features to understand - and then minimise- how much data your queries have to trawl through.
- Column Codecs are a slept-on feature, smart use of them can shrink your storage size, often with positive effects on your query latency in-turn.
- Write in database-friendly format!

Questions

Thank you for listening!

ClickHouse