Synq ❤ ClickHouse

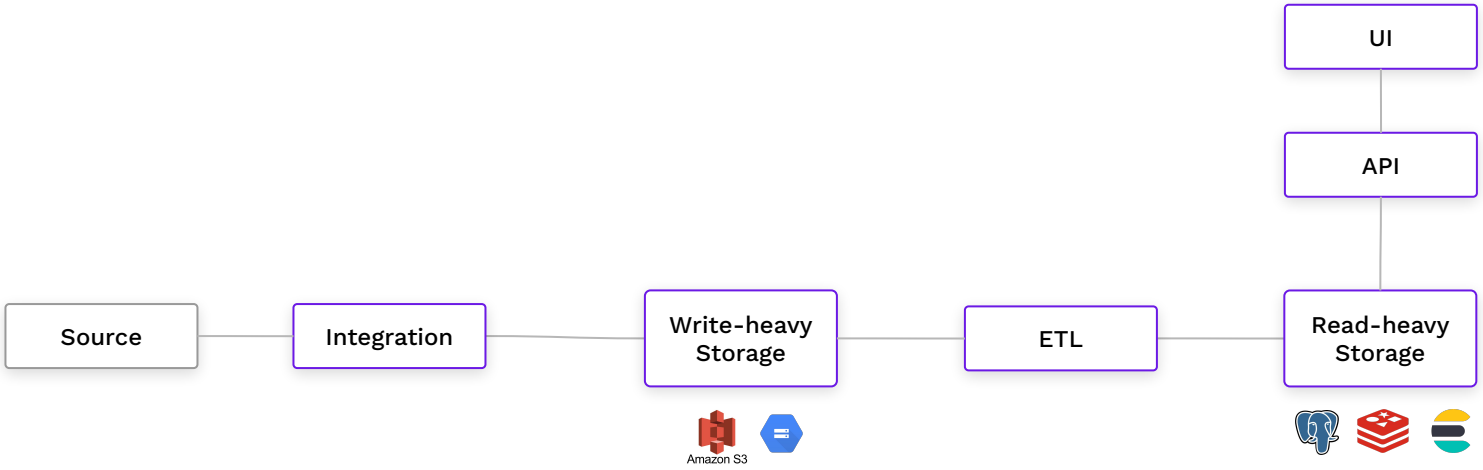
Building user-facing and internal applications with ClickHouse

Hi, I am Petr! /ˈpiːtə(r)/

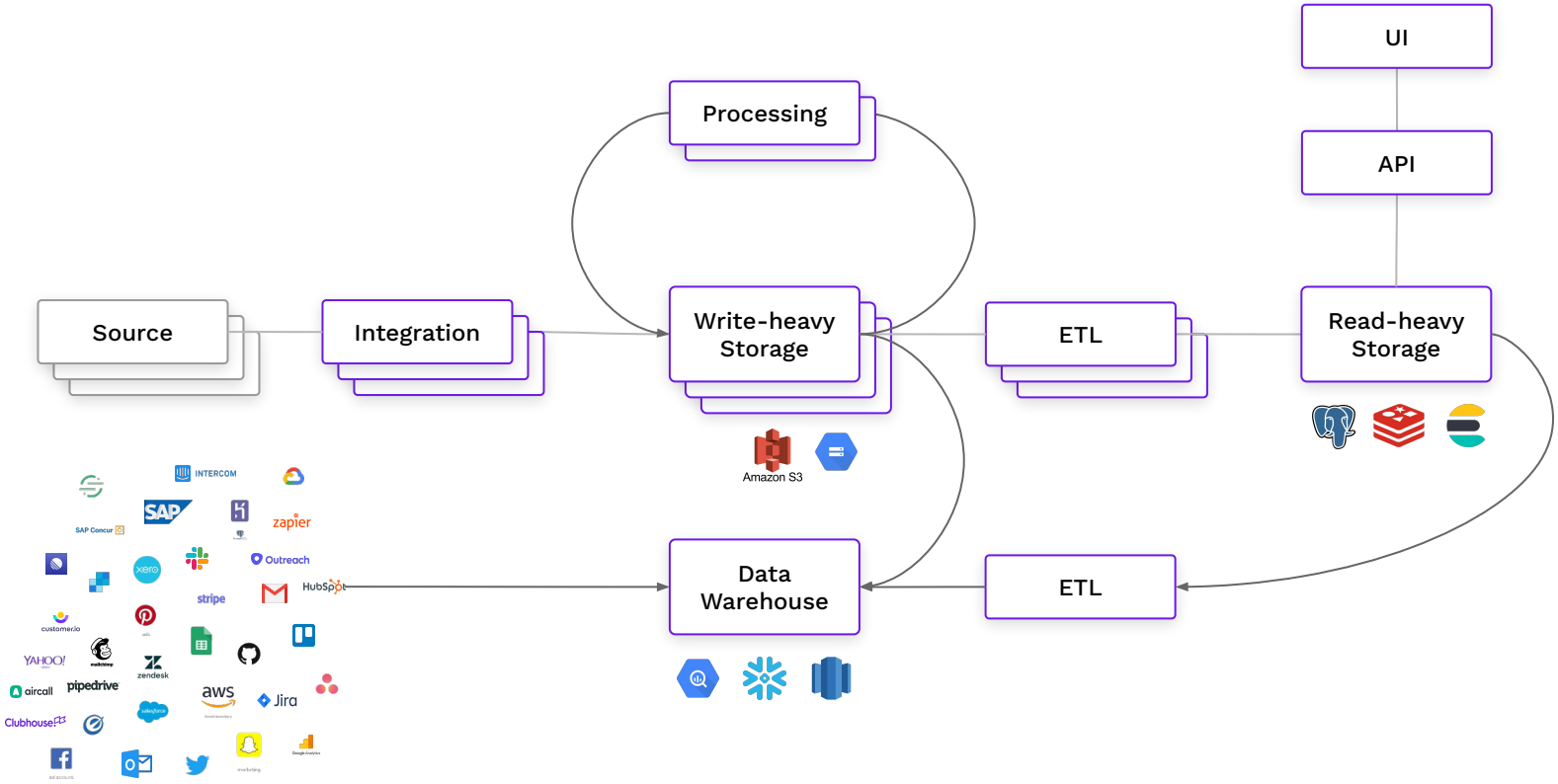
Engineer at heart.
Product, data geek.
Founder / Synq.

ex-[Pleo/GWI/CEAi] (=data intensive businesses)

Common design



Common design



What’s difficult?

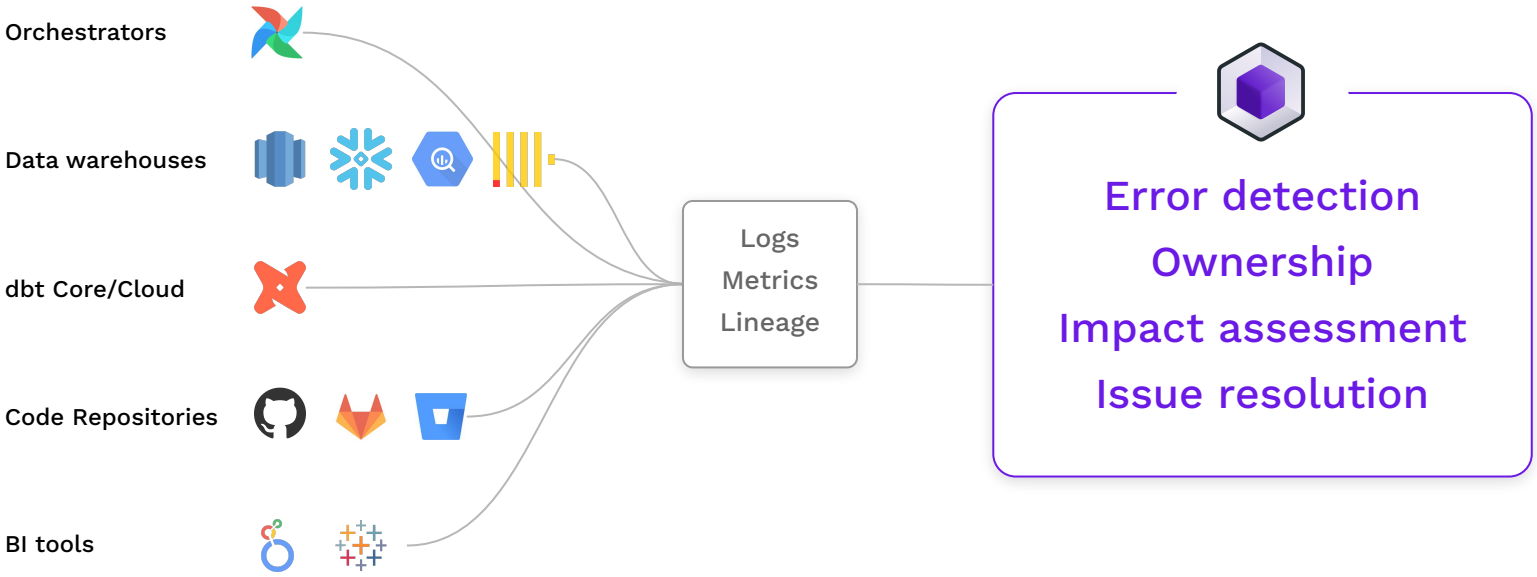
Lots of specialised storages.

Lots of code to move data around.

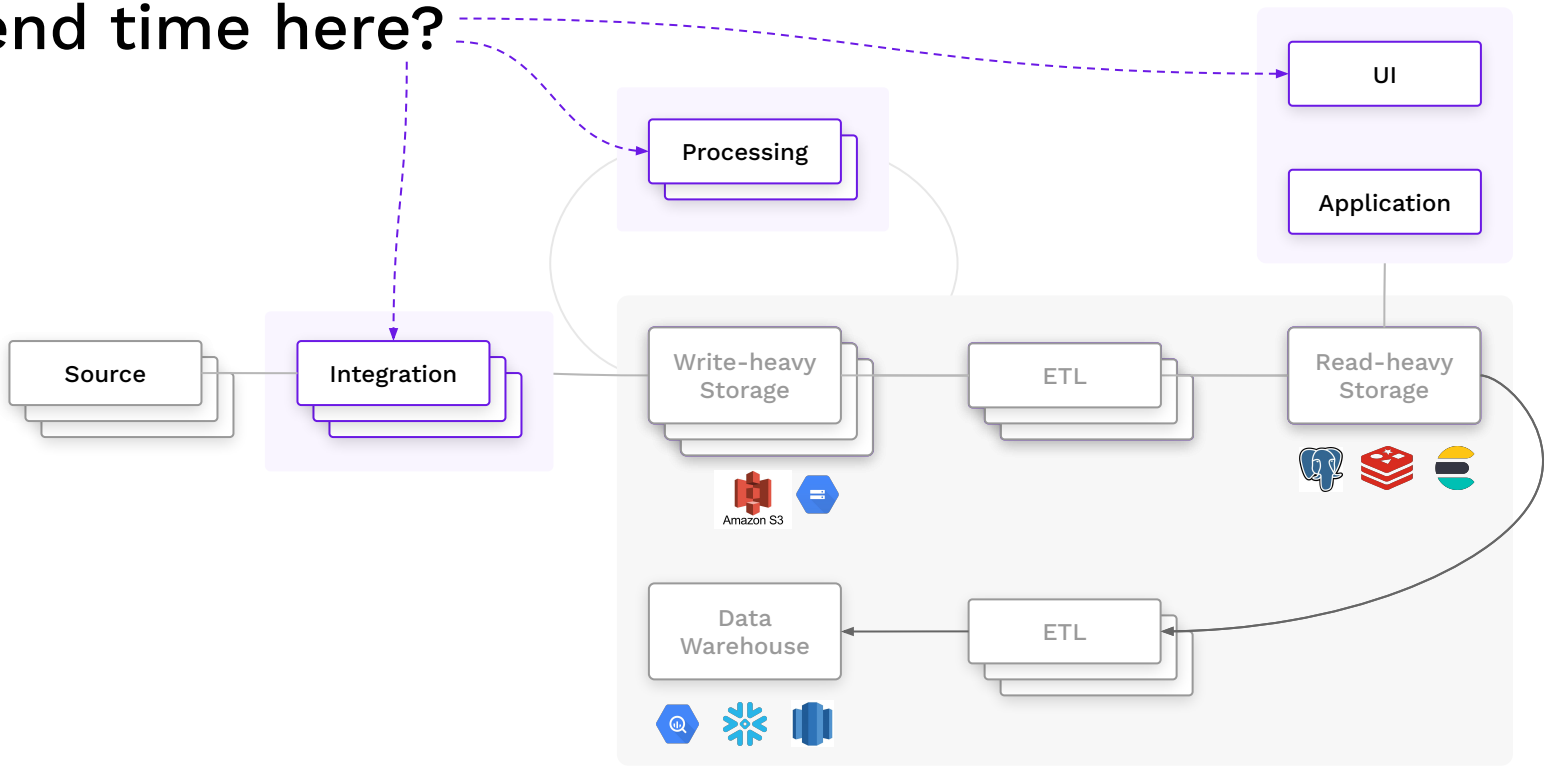
Lots of boilerplate.

How can we build data intensive applications fast(er)?

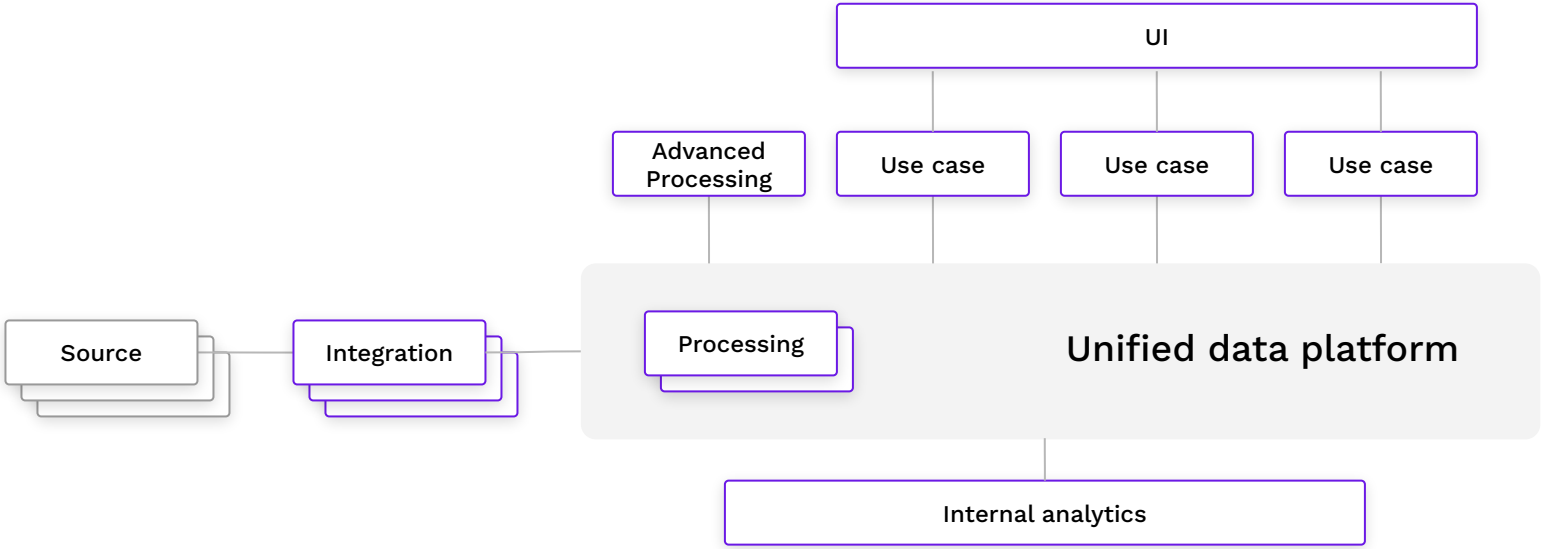
Synq, a data observability platform



And how can we spend time here?



Can we do this?



Requirements

Scalable ingest

Ingest that scales, so we can ingest versatile data and build near-real time features.

Metric
100ks records/s
“Burst-friendly”

Flexible processing

Processing capabilities to create new formats of data, so we can develop new features fast.

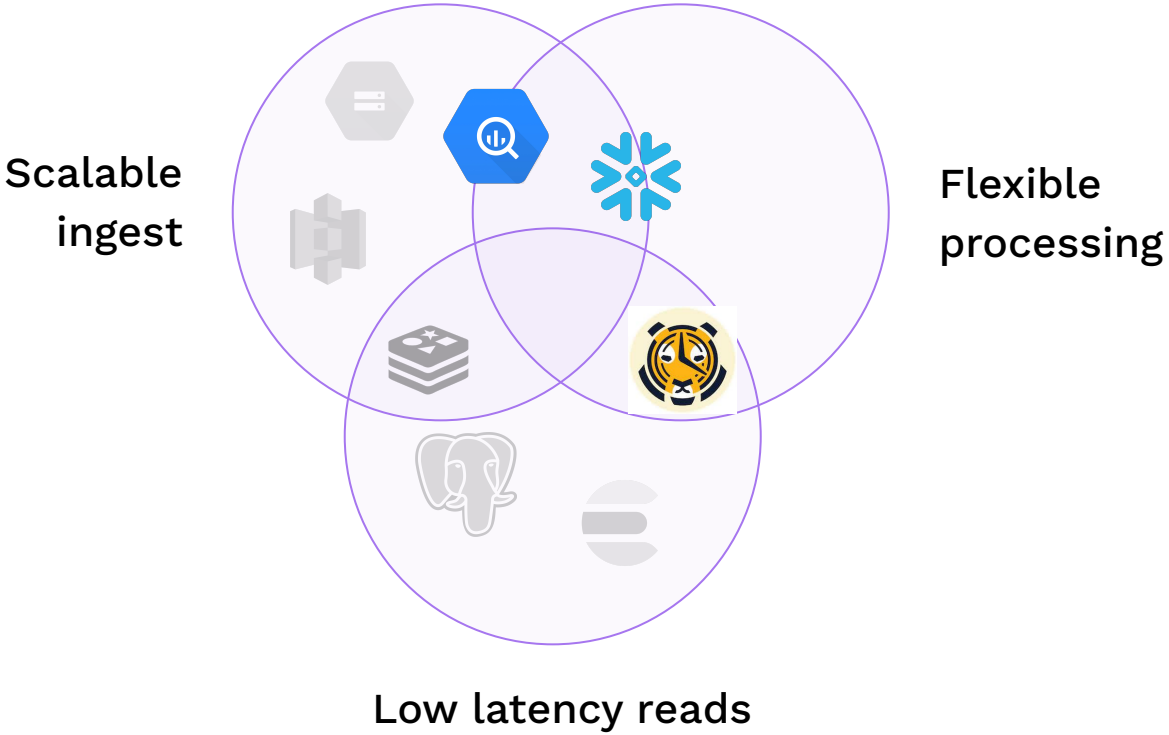
Metric
Time-to-prototype
Reliability

Low latency reads

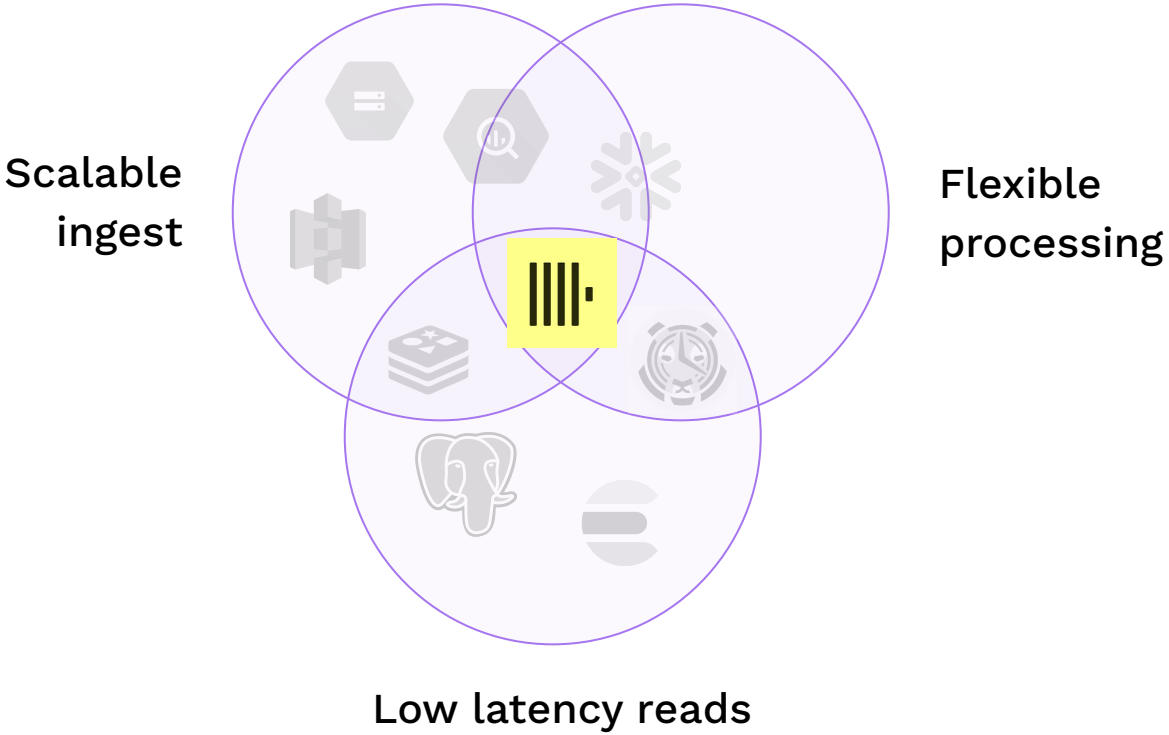
User-friendly query latency for specific data, so we can serve user-facing use cases.

Metric
100ms latency
Handles complex queries

These requirements are hard



These requirements are hard



Bonuses

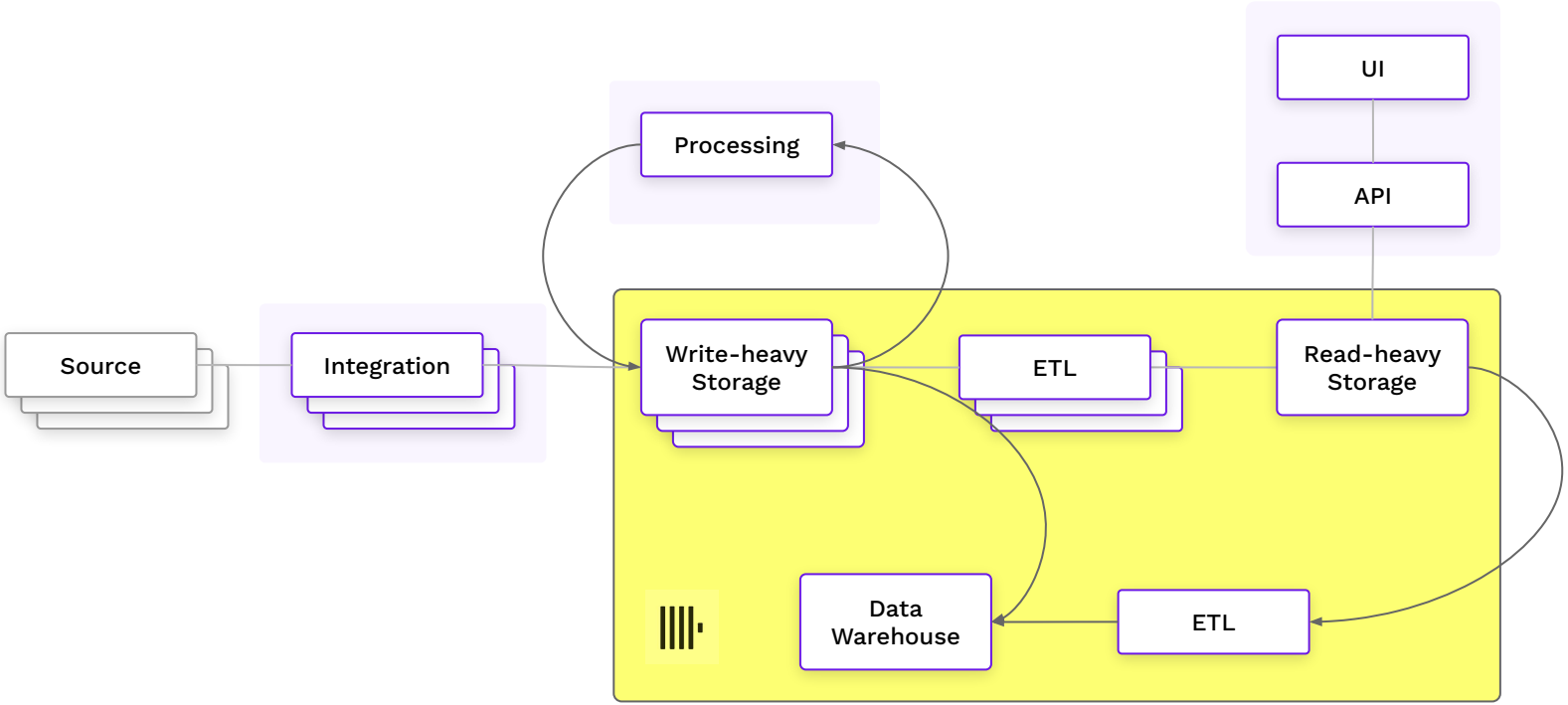
—Runs locally (=easy to test)

—Resilient (=hard to break)

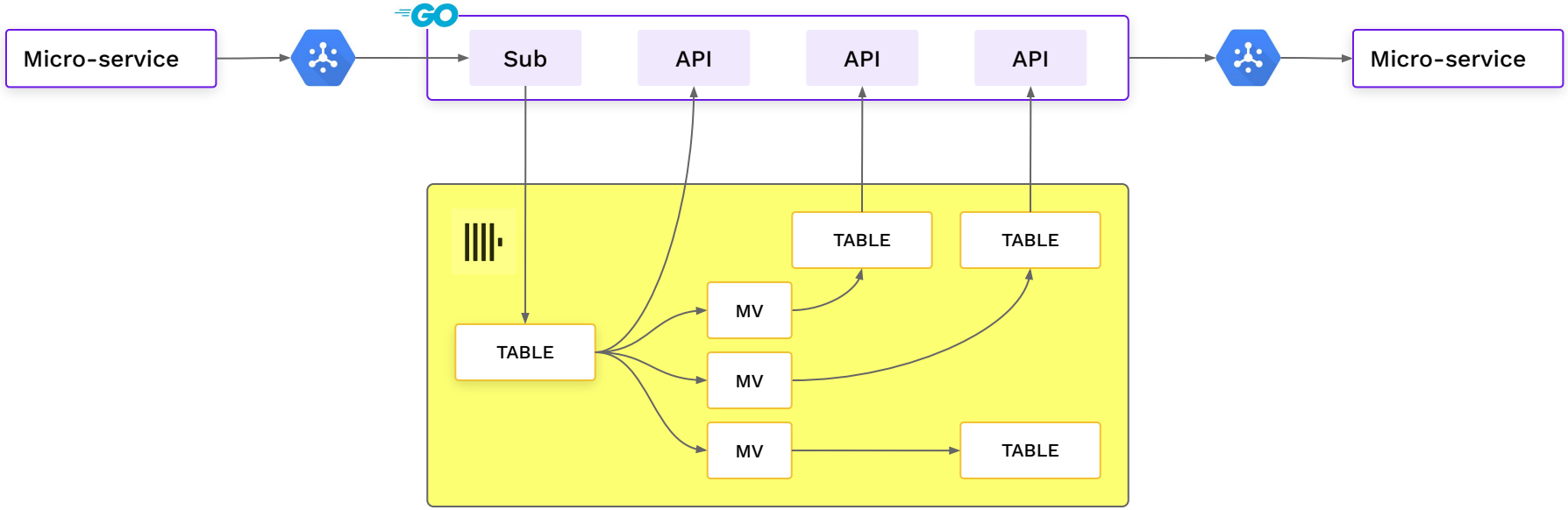
—Cost efficient

Building with ClickHouse

Pushing complexity “down” to ClickHouse



Anatomy of micro-service



Ingest

```
CREATE TABLE IF NOT EXISTS runs (
    workspace String
    , id String
    …
    , run_status Int32
    , message String
    , created_at DateTime64(8, 'UTC')
    , started_at DateTime64(8, 'UTC')
    , finished_at DateTime64(8, 'UTC')
    , meta String,
    INDEX bf_target(target)
      TYPE bloom_filter GRANULARITY 3
) Engine = ReplacingMergeTree()
    ORDER BY (workspace, id, created_at)
    SETTINGS index_granularity=2048
;
```

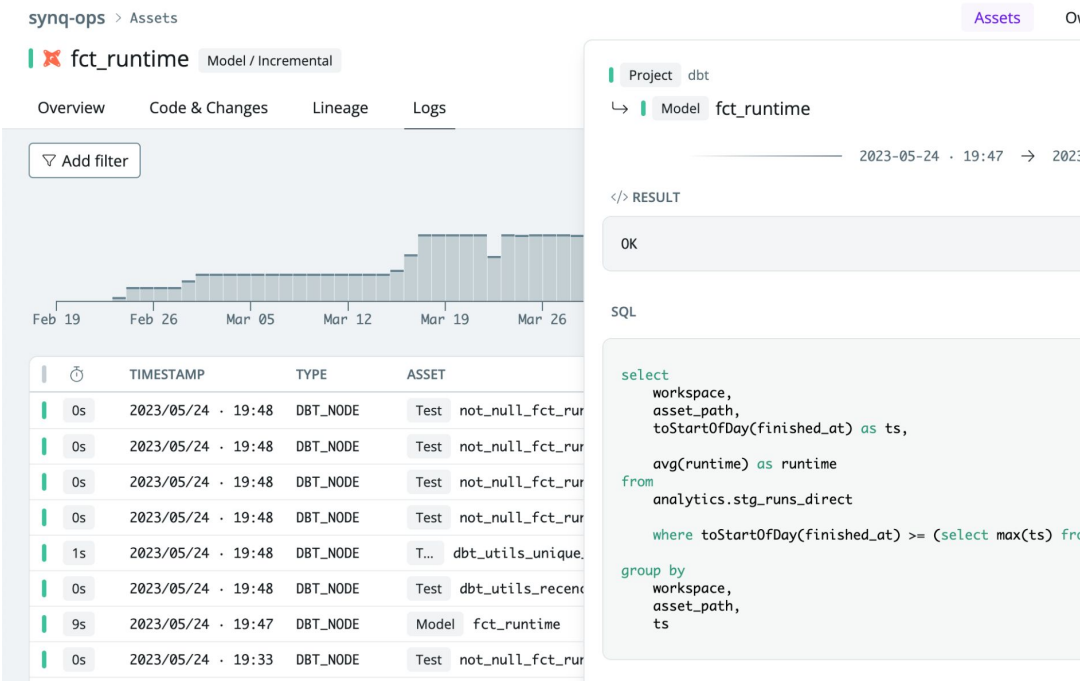
```go
func (api *RunApiImpl) AddRunAsync(
    ctx context.Context, conn driver.Conn, run *corev1.Run
) error {
  sql, args, err := buildRunQuery(run)
  query, err := clickhouse_synq.Bind(time.UTC, sql, args...)

  ctx = clickhouse.Context(ctx, clickhouse.WithSettings(
    clickhouse.Settings{
      "async_insert_max_data_size":   "100000000",
      "async_insert_busy_timeout_ms": "10000",
    }
  ))

  return conn.AsyncInsert(ctx, query, false)
}
```

```
+--------+-----+------+---------+------------+----------+
|table   |parts|marks |rows     |uncompressed|compressed|
+--------+-----+------+---------+------------+----------+
|runs    |381  |114918|228515154|691.34 GiB  |58.24 GiB |
+--------+-----+------+---------+------------+----------+
```

https://clickhouse.com/docs/en/operations/settings/settings#async-insert

Logs

```
SELECT
  workspace,
  id,
  parent_ids,
  …
  created_at,
  started_at,
  finished_at,
  meta
FROM
  runs
WHERE
  workspace = $1
  AND id IN [$2]
ORDER BY
  created_at DESC
LIMIT
  1 BY id
```



Asset Lists (with MVs)

```
CREATE TABLE IF NOT EXISTS statuses (
  workspace String,
  target String,
  run_status Int32,
  created_at DateTime64(8, 'UTC')
)
Engine = ReplacingMergeTree()
ORDER BY (
  workspace,
  target,
  created_at);

CREATE MATERIALIZED VIEW IF NOT EXISTS
statuses_mv TO statuses
AS
  SELECT
    workspace,
    target,
    run_status,
    created_at
  FROM runs ARRAY JOIN target;
```

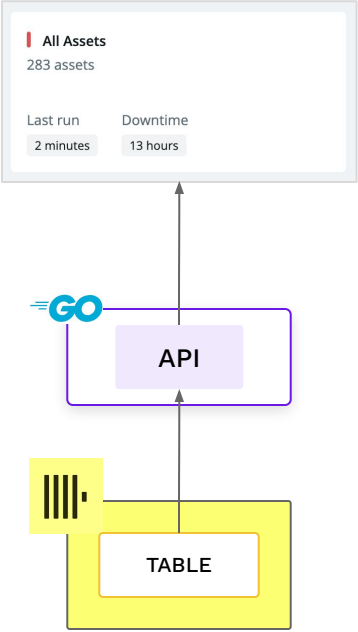
```
WITH deduplicated AS (
  select
    target,
    target_hash,
    run_status,
    created_at
  from
    statuses
  where
    workspace = $1
    and target IN ($2)
    and created_at BETWEEN …
  limit 1 by
    target,
    target_hash,
    created_at
)

, gaps_islands AS (
  …
)
```

	TYPE	NAME
	Monitor	latest_commits freshne
	Monitor	runs volume
	Monitor	runs freshness
	Monitor	statuses volume
	Monitor	statuses freshness
	Monitor	statuses_direct volume
	Monitor	statuses_direct freshne
	Monitor	alerts_prepared_alerts_
	Monitor	alerts_prepared_alerts_
	Monitor	core_assets_processed
	Monitor	core_assets_processed
	Monitor	pages volume
	Monitor	pages freshness
	Monitor	users volume
	Monitor	users freshness
	Monitor	asynchronous_insert_lo
	Monitor	asynchronous_insert_lo

table	rows	uncompressed	compressed
statuses	**653,904,032**	**72.09 GiB**	**3.05 GiB**
runs	228,515,154	691.34 GiB	58.24 GiB

Performance is 🔥

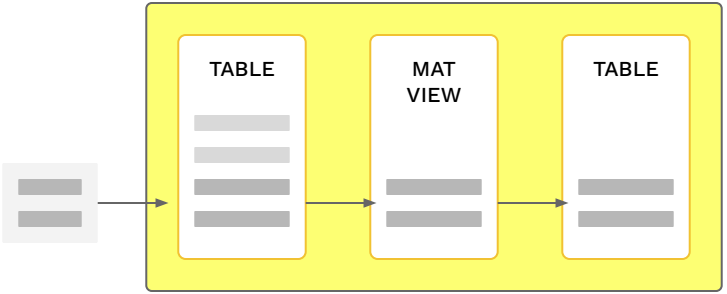
NAME	REQUESTS	AVG LATENCY
/runs.runs.v1.RunsService/BatchRuns	10.6k	46.9 ms
/runs.statuses.v1.StatusesService/BatchLatestDirectStatus	3.13k	65.1 ms
/runs.statuses.v1.StatusesService/BatchStatusesByLatest	1.22k	217 ms
/runs.commits.v1.CommitsService/BatchLatestCommits	544	31.9 ms
/runs.statuses.v1.StatusesService/ListPathsByStatuses	337	152 ms
/runs.runs.v1.RunsService/GetAggregatedMetrics	245	153 ms
/runs.statuses.v1.StatusesService/BatchLastStatuses	87	91.3 ms
/runs.runs.v1.RunsService/BatchLatestRun	66	109 ms



In-app analytics: MVs limitations

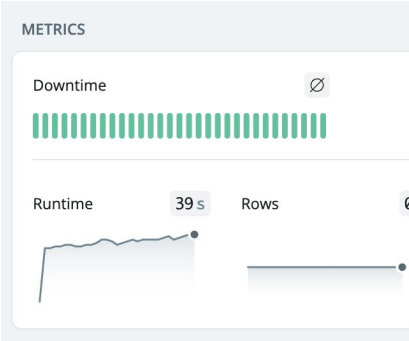

```
CREATE MATERIALIZED VIEW … AS
  SELECT
    …,
    run_status,
    any(run_status) over (
      partition by …
      order by finished_at
      rows between 1 preceding and
      1 preceding
    ) AS last_run_status
  FROM
    runs
;
```

In-app analytics: MVs limitations

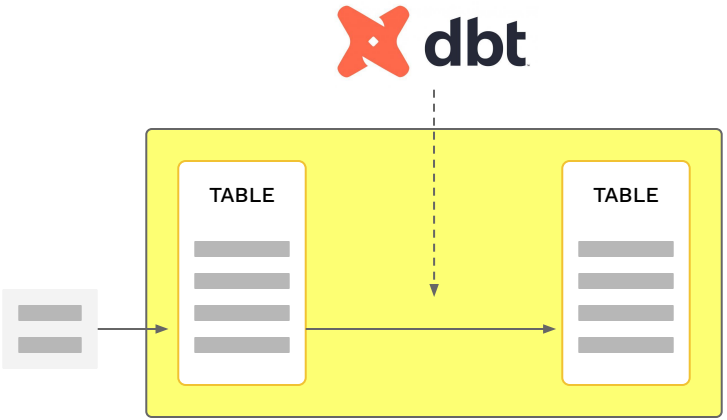


Materialized views don't query source tables, they see only new block of data.

```
CREATE MATERIALIZED VIEW … AS
  SELECT
    …,
    run_status,
    any(run_status) over (
      partition by …
      order by finished_at
      rows between 1 preceding
        and 1 preceding
    ) AS last_run_status
  FROM
    runs
;
```

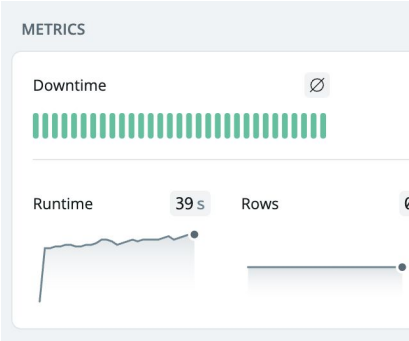


In-app analytics: Introducing dbt

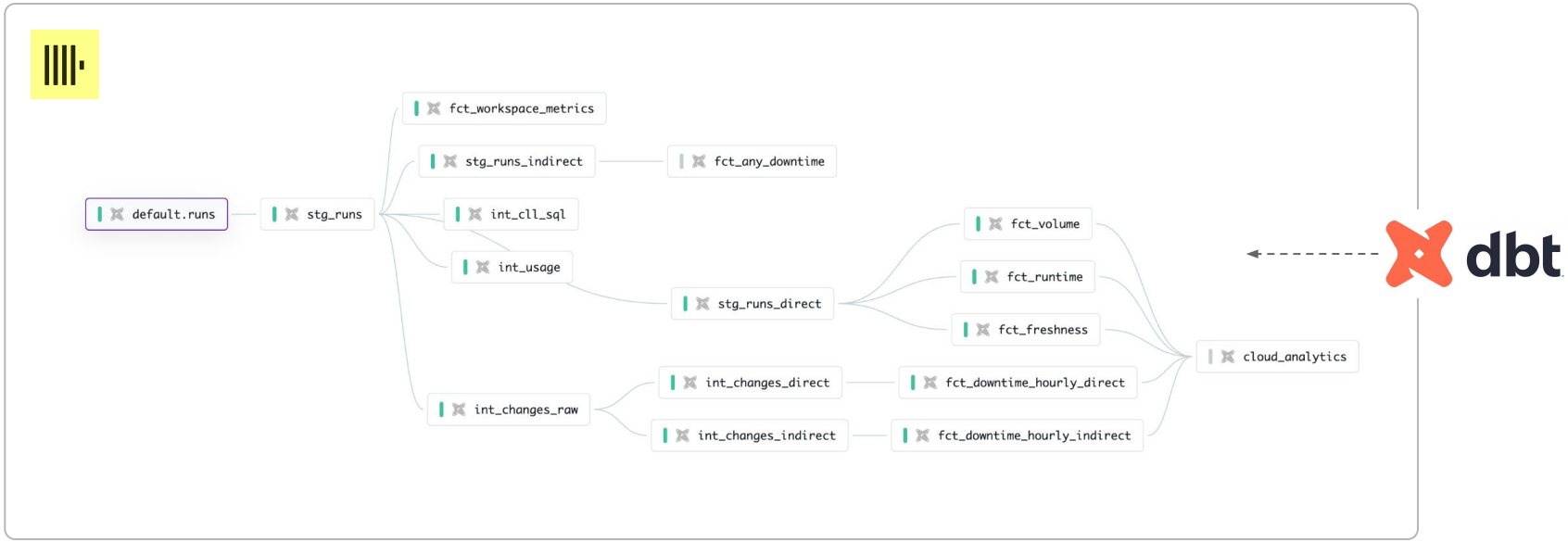


Materialized views don't query source tables, they see only new block of data.

```
CREATE TABLE … AS
  SELECT
    …,
    run_status,
    any(run_status) over (
      partition by …
      order by finished_at
      rows between 1 preceding
        and 1 preceding
    ) AS last_run_status
  FROM
    runs
;
```



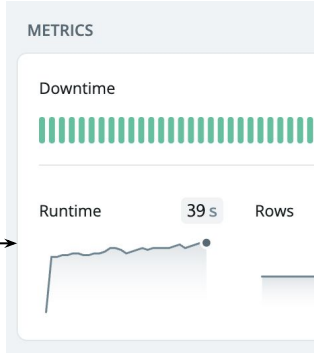
Introducing dbt: DAG



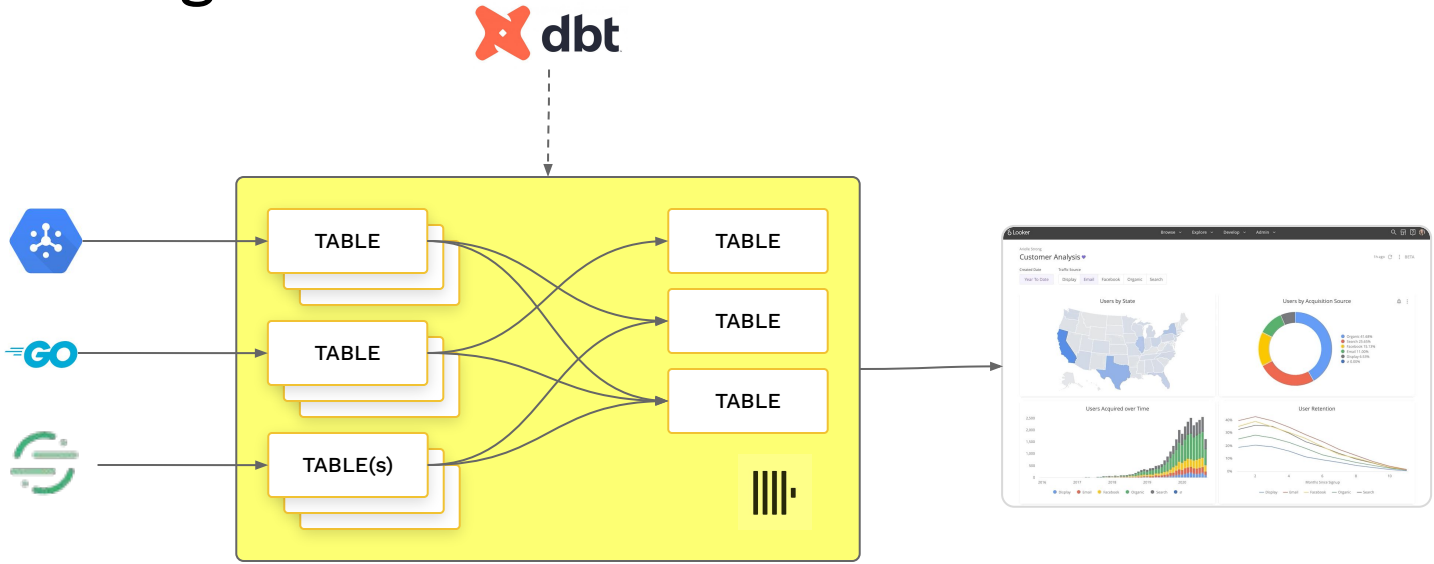
Introducing dbt: Models + queries

```
{{ config(
    order_by='(workspace, asset_path, ts)',
    engine='ReplacingMergeTree()',
    materialized='incremental',
    unique_key=[
      'workspace', 'asset_path', 'ts'
    ]
)}}

select
    workspace,
    asset_path,
    toStartOfDay(finished_at) as ts,
    avg(runtime) as runtime
from
    …
```

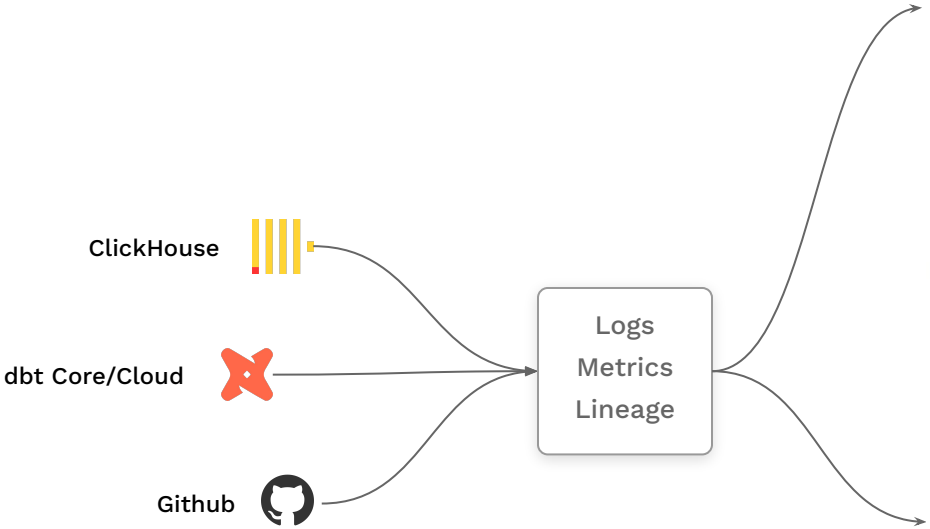
```
select
  asset_path as path,
  ts,
  runtime as value
from
  analytics.fct_runtime
where
  workspace = $1
  and asset_path IN [$2]
  and ts > …
```



Adding data warehousing



Monitoring ClickHouse



Synq APP 10:10 AM

dbt Project **dbt**

```
Invocation of `dbt build`
```

1 new issues detected:

@Petr Janda 🔥 IMPORTANT dbt model **fct_downtime_hourly_indirect**

```
Database Error in model fct_downtime_hourly_indirect
(models/product/fct_downtime_hourly_indirect.sql)
  Code: 241.
  DB::Exception: Memory limit (total) exceeded: would use 14.41 GiB (attempt
to allocate chunk of 70254624 bytes), maximum: 14.40 GiB. OvercommitTracker
decision: Query was selected to sto…
```

See more

📝 last change: **add segment, refactoring main models** by Petr Janda *(1d ago)*, # impacted assets: **1**

Synq APP 5:02 PM

Anomaly detection **clickhouse**

```
Executed 4 monitors, 1 failed.
```

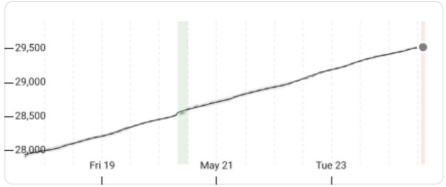
1 new issues detected:

@Petr Janda clickhouse table **alerts_prepared_alerts_v1** triggered by Monitor **alerts_prepared_alerts_v1 volume**

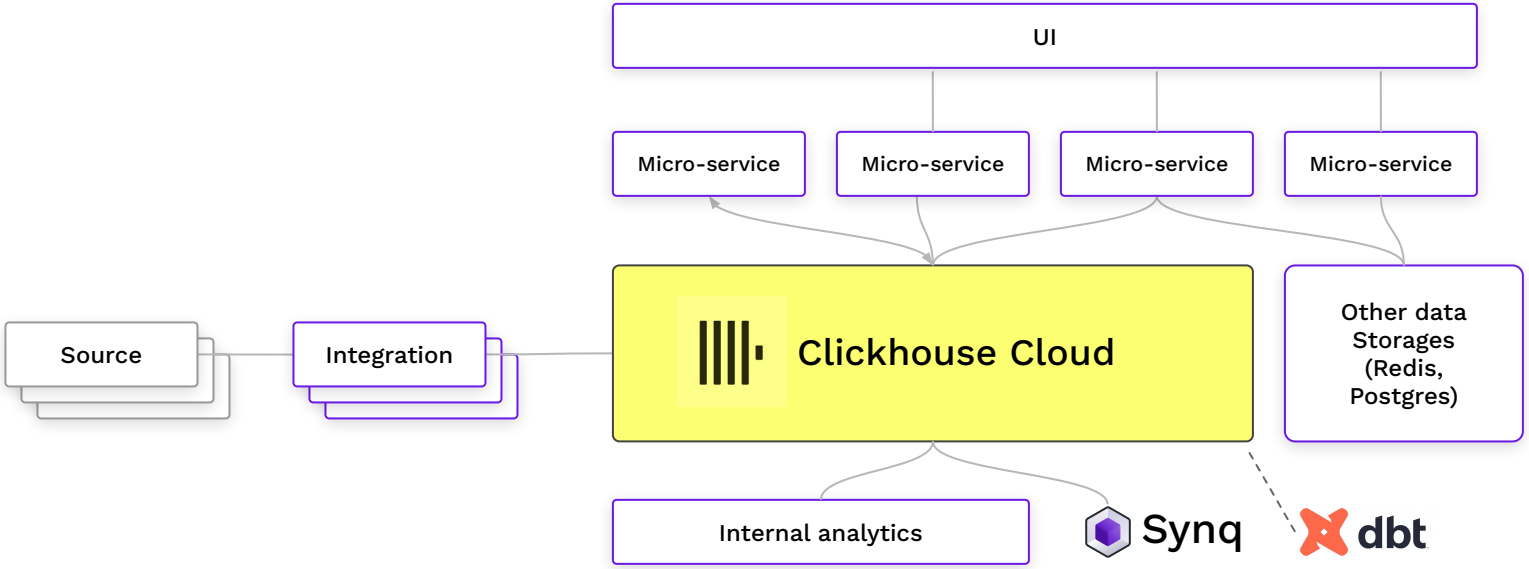
```
Anomaly detected after 3 observations: Value dropped. Expected 29541, was
29520.
```

📝 # impacted assets: **3**

alerts_prepared_alerts_v1 (10 kB)



“No boilerplate” system



Why ClickHouse rocks 🤘

Scalable ingest

We load data to ClickHouse in a fire and forget way. After tuning thresholds of Async Inserts it just takes it.

Metric

100ks records/s ✔️

“Burst-friendly” ✔️

Flexible processing

Data for new features can be prototyped in SQL in 1-2 hours. Perfect for a small startup team.

Metric

Time-to-prototype ✔️

Reliability ✔️

Low latency reads

All queries are either already fast or we have plenty of room to optimise them. Even the complex ones.

Metric

100ms latency ✔️

Handles complex queries ✔️

Thank you!

Email me at **petr@synq.io**.

Sign up at **synq.io**.

DM me at @LinkedIn.