Data Contracts in the Modern Data Stack

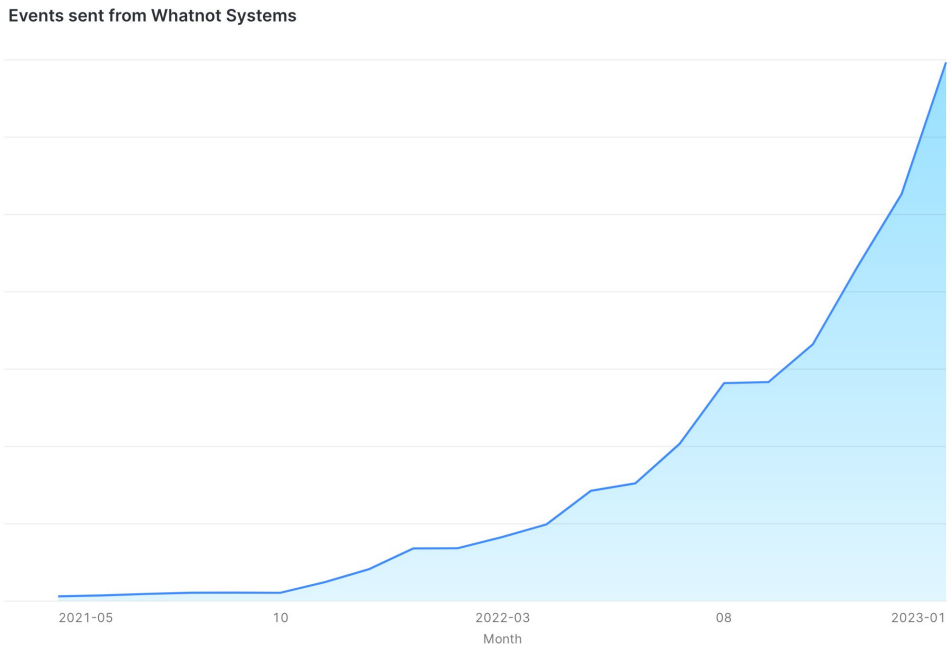
Zack Klein  whatnot

What is Whatnot?

- Livestream & async e-commerce platform
- Collectibles and community-driven markets
- Fastest growing marketplace in the United States.

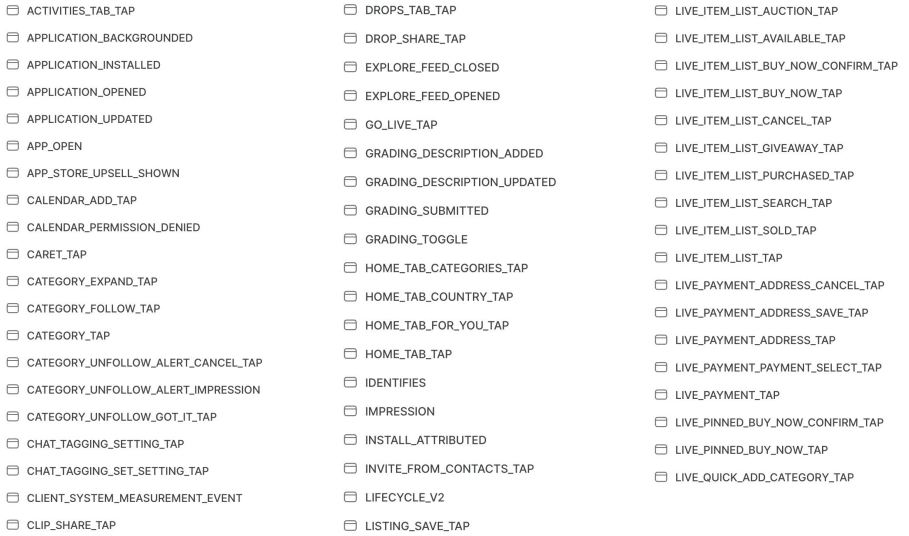
Our data stack

- We write a lot – check out our blog
- Data stack v3 – Snowflake, AWS, Dagster, DBT, Segment, Kafka.
- Started ~2 years ago when the company was ~20 employees. Now we are ~400!
- Today we'll be talking about **data contracts in our events** system.
- Over the past ~2 years, the number of events we've sent per month has increased **over 100x**, with an average MoM increase of **~30%**



The problems

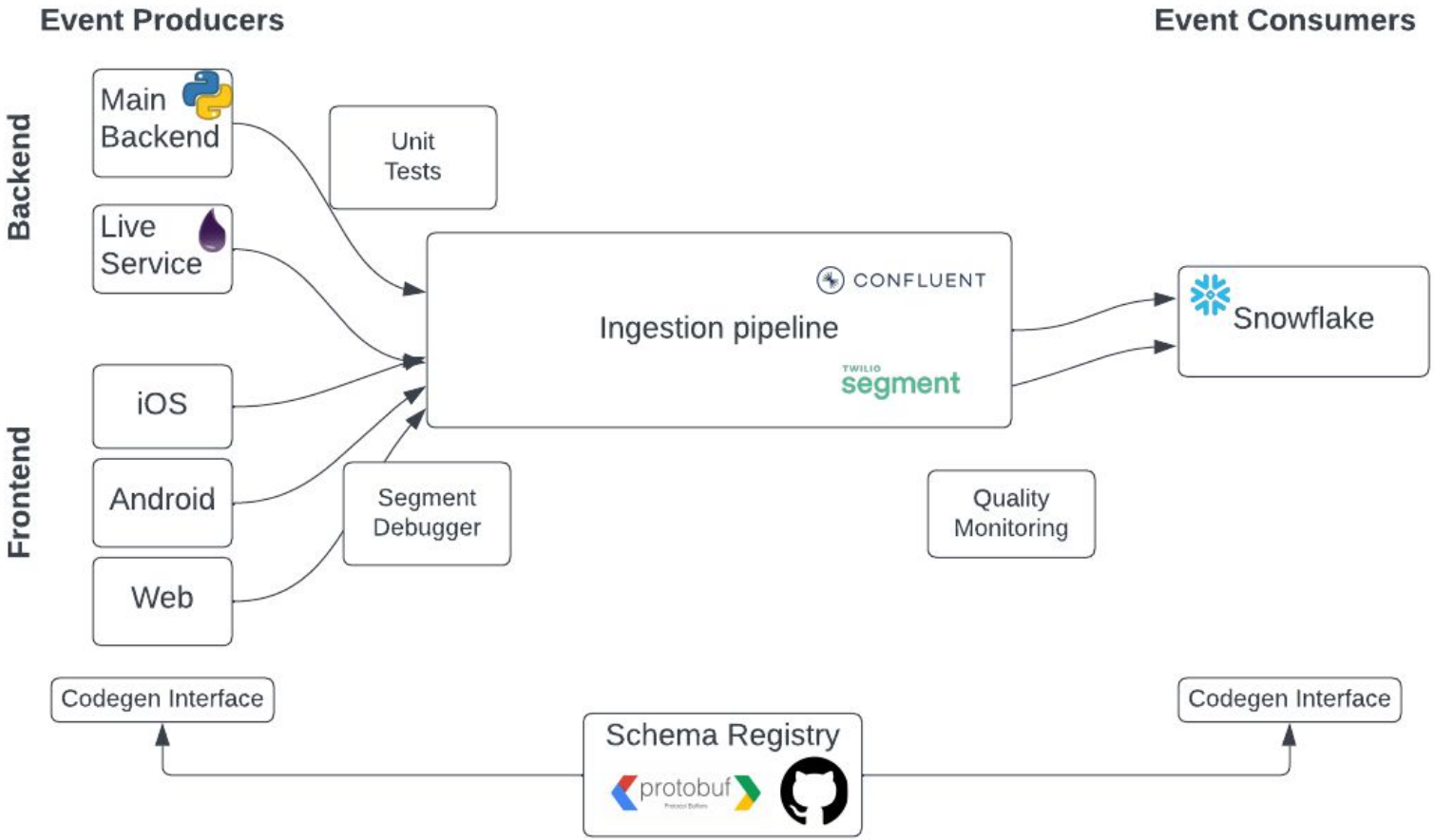
- Major problems:
  - Wild west – inconsistent/huge number of tables, fields, etc.
  - Starting from scratch each time we wanted to make new events.
  - No ownership/accountability – difficult and slow to fix issues.
- Whatnot is super data-driven – we saw this as a big risk, and decided to invest heavily to improve it.



What we did

A complete overhaul of the events system

- Significant investment in the unification of our events into one “data highway”.
- Conformed all existing events into *Actor Action Object* model
- We accomplished this by building 4 components:
  - Interface
  - Schema
  - Pipeline
  - Exposure
- Leaned heavily on code generation and PR review process to allow us to move fast with high quality.


Event Producers
Event Consumers
Backend
Frontend
Main Backend
Live Service
iOS
Android
Web
Unit Tests
Segment Debugger
CONFLUENT
Ingestion pipeline
TWILIO segment
Snowflake
Quality Monitoring
Codegen Interface
Schema Registry
protobuf
Codegen Interface

Example: a user follows another user

Step 1: Declare schema

```
message UserFollowedUserEvent {
  google.protobuf.Int64Value follower_id = 1;
  google.protobuf.Int64Value followee_id = 2;
}
```

```
UserFollowedUserEvent user_followed_user = 17 [
  (event_metadata).description.value = "Fires when a user starts following other user.",
  (event_metadata).team_owner.value = "Foundations"
];
```

Step 2: Implement producer

```
def user_followed_user(
    *,
    follower_id: int,
    followee_id: int,
):
    event = UserFollowedUserEvent()
    set_nullable_value(event.follower_id, follower_id)
    set_nullable_value(event.followee_id, followee_id)
```

```
user_followed_user(follower_id=user_id, followee_id=user_id_to_follow)
```

Shown here: Protobuf Python API implementation

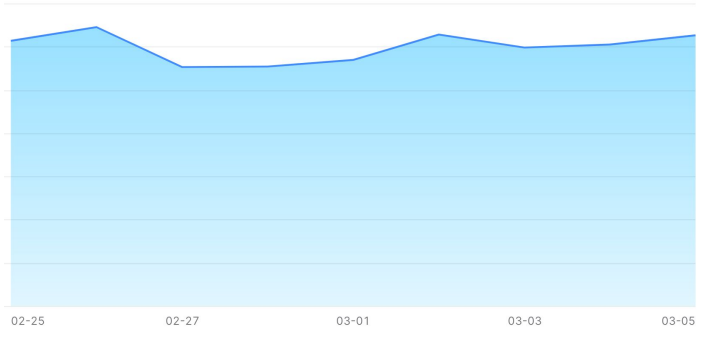
Step 3: Query

BACKEND__EVENTS

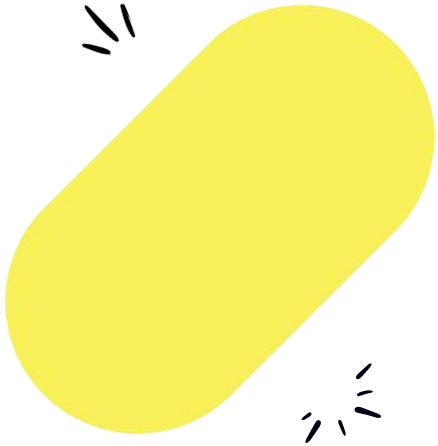
FRONTEND__EVENTS

```sql
select *
from backend__events
where event_name = 'user_followed_user'
limit 10;
```

```sql
select date_trunc(day, event_timestamp) dt
    , count(*)
from backend__events
where 1=1
and event_name = 'user_followed_user'
and event_timestamp >= '2023-02-25'
and event_timestamp < sysdate()::date
group by 1
order by 1 desc;
```



Maintaining quality



We catch errors at various stages of development:

- **Before anything is implemented**
  - Automated checks (using Python unit tests and open source protobuf tooling) on the schema to ensure backwards compatibility, schema validity, naming conventions, etc..
- **During development**
  - Easy path to write unit tests asserting semantic quality of the data.
- **After we've shipped**
  - In-flight monitoring using DataDog metrics and alerting on schema issues.
  - QA checks that run in the warehouse using custom observability tooling.

Learnings

- Code generation is A Good Thing.
- The earlier the better.
- The more focus on the users of the system (event producers and the data consumers), the better.

Thank you!