CODE GENERATION WITH PROTOBUF @ CANVA

Wade Jensen
Arthur Baudry

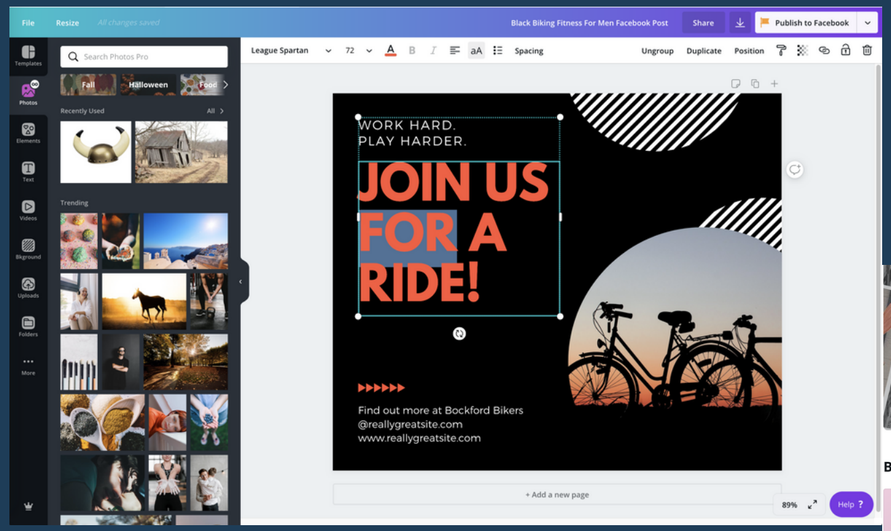
@wadejensen

@getwaded

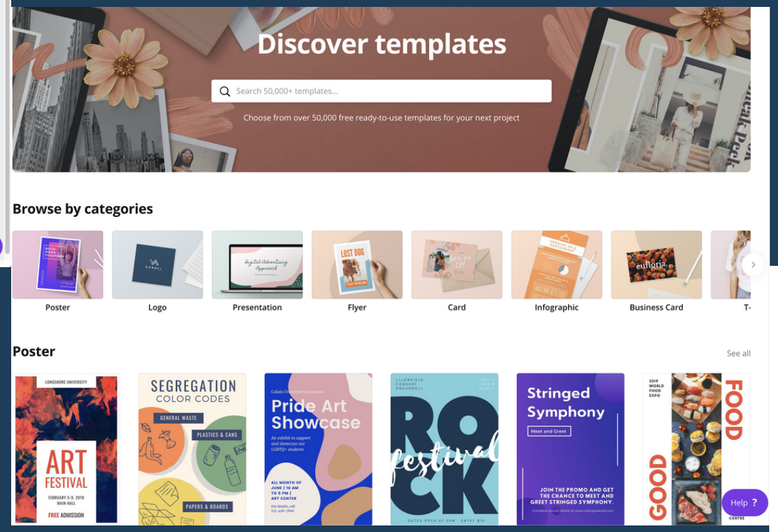
@ArthurBaudry

@ArthBaud

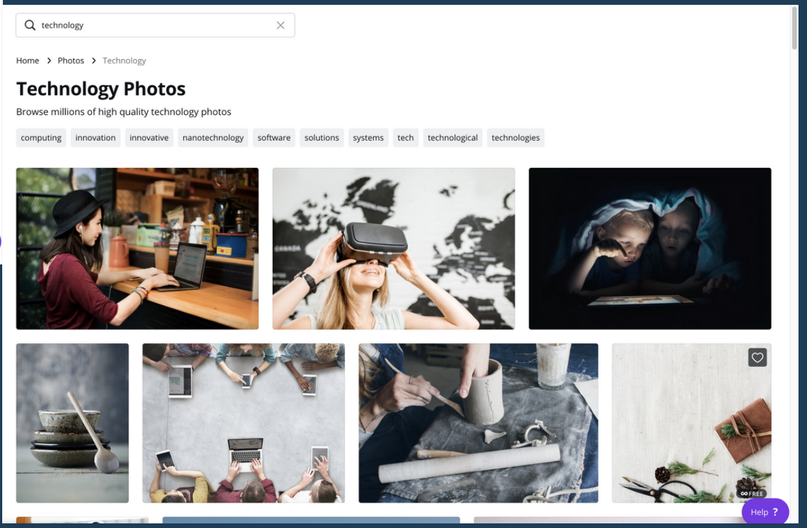
Editor



Templates



Photos



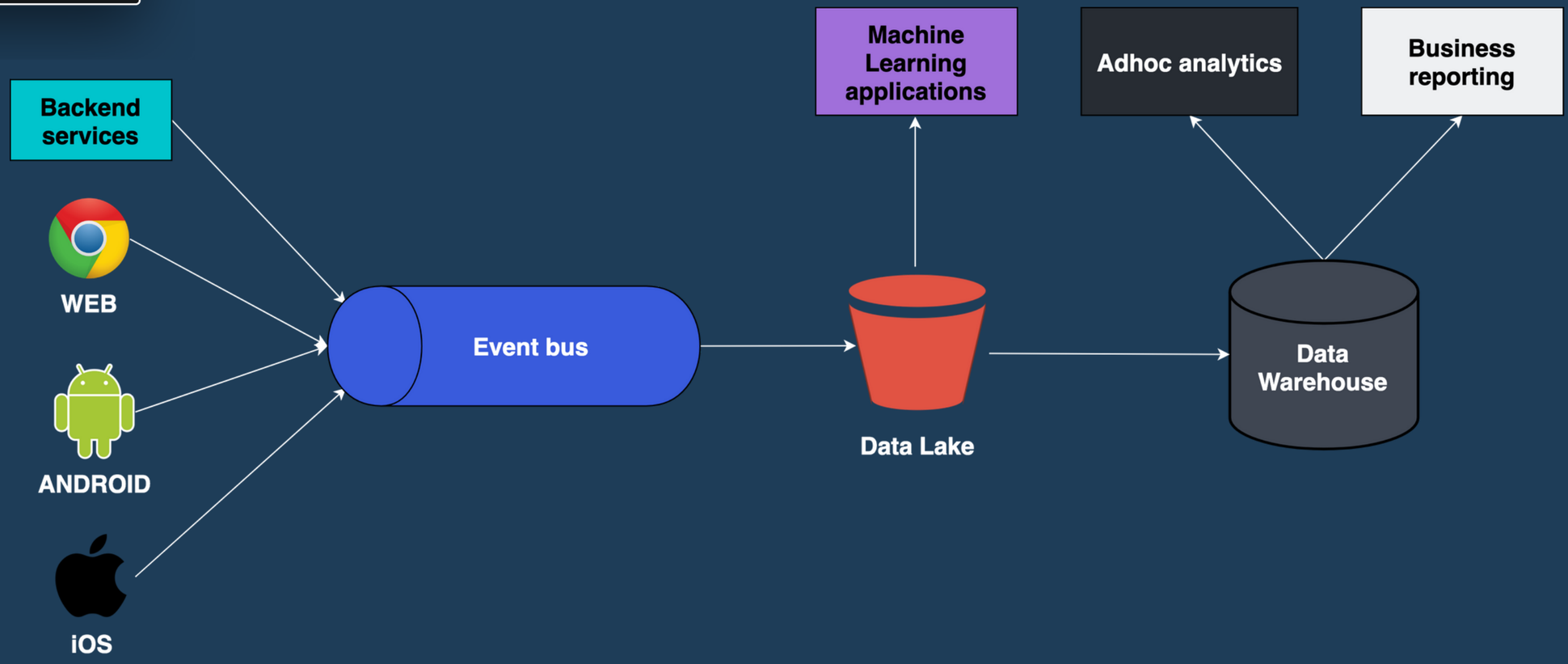
DATA ENG TEAM

Help Canva make better decisions with data

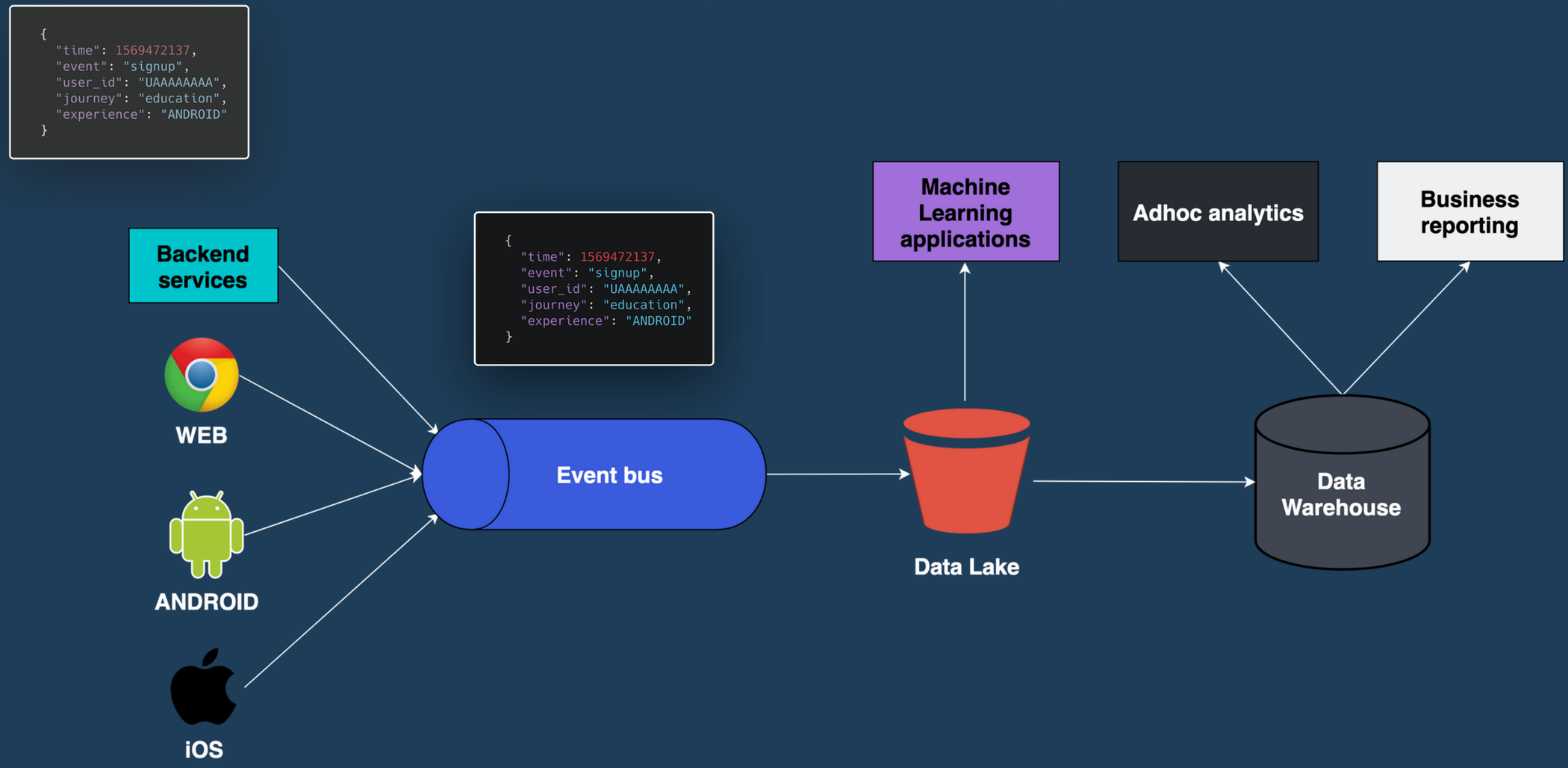
- Make data easy to produce
- Make data easy to consume
- Data storage and management
- Platform for data pipelines

First party event ingestion

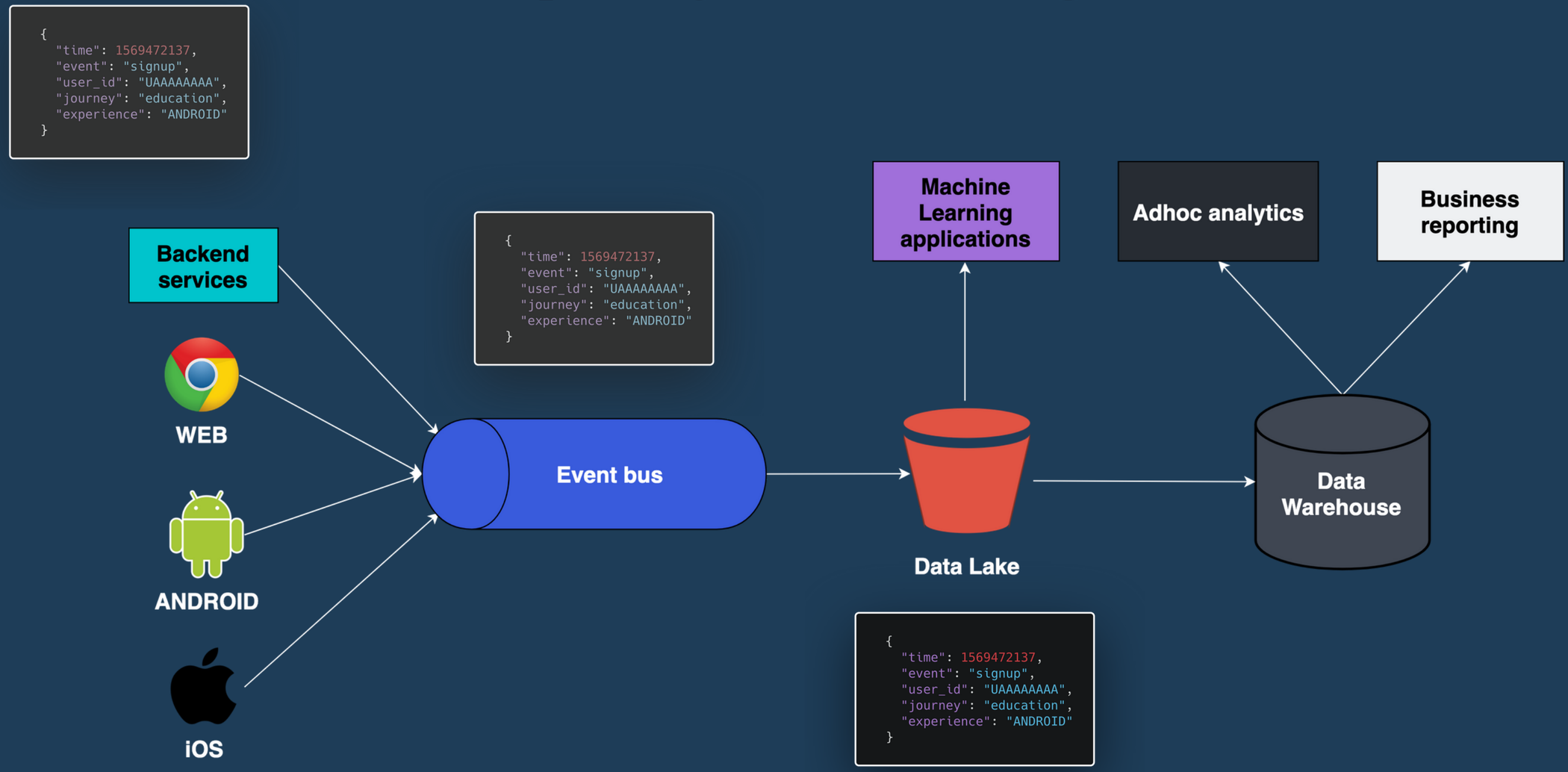

```
{
  "time": 1569472137,
  "event": "signup",
  "user_id": "UAAAAAAAA",
  "journey": "education",
  "experience": "ANDROID"
}
```

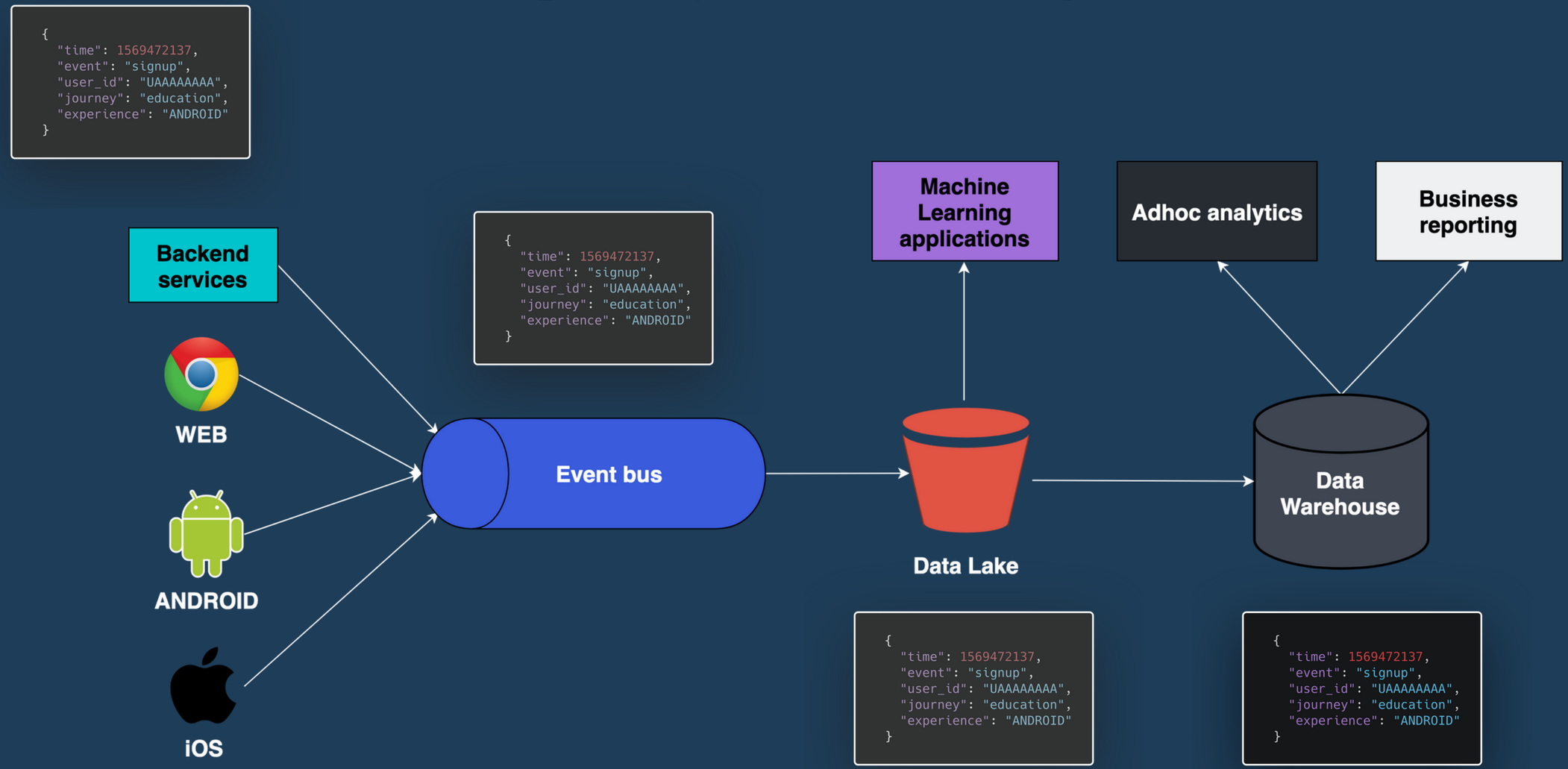
First party event ingestion



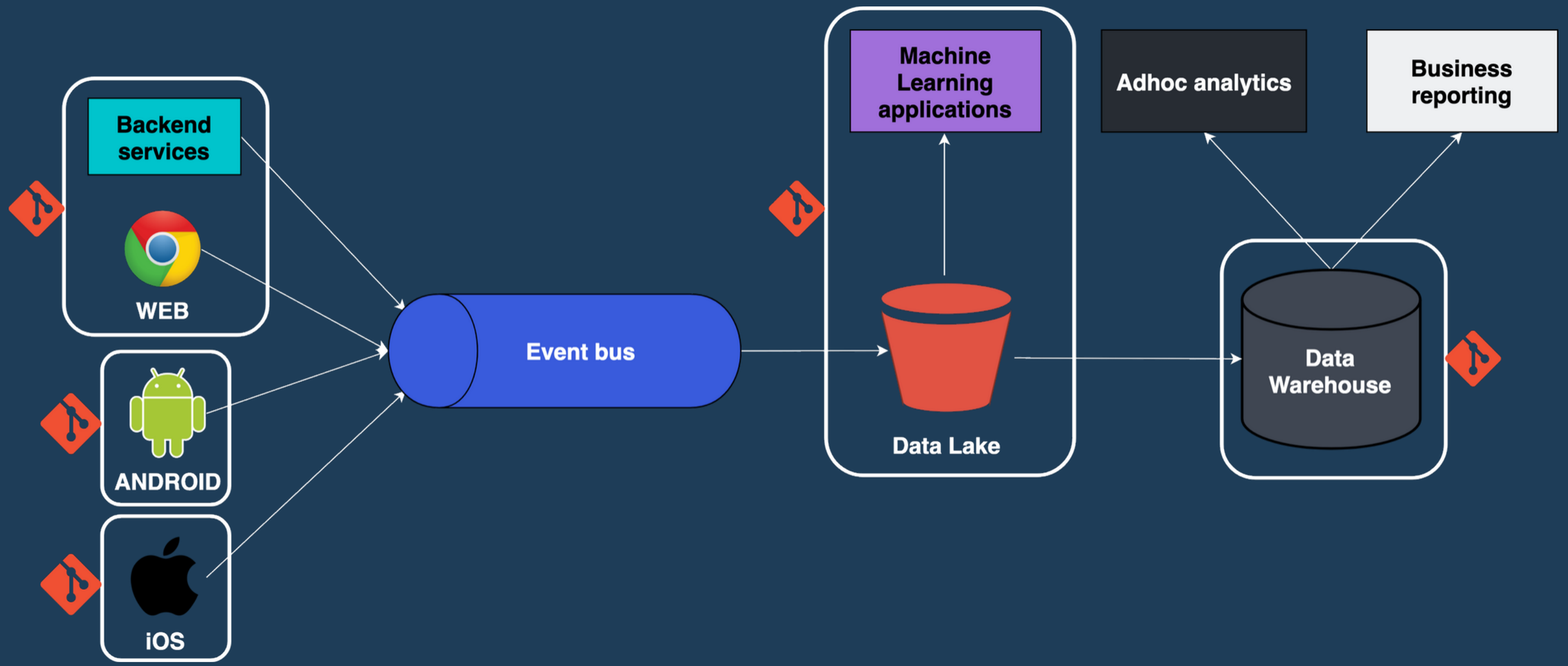
First party event ingestion



First party event ingestion

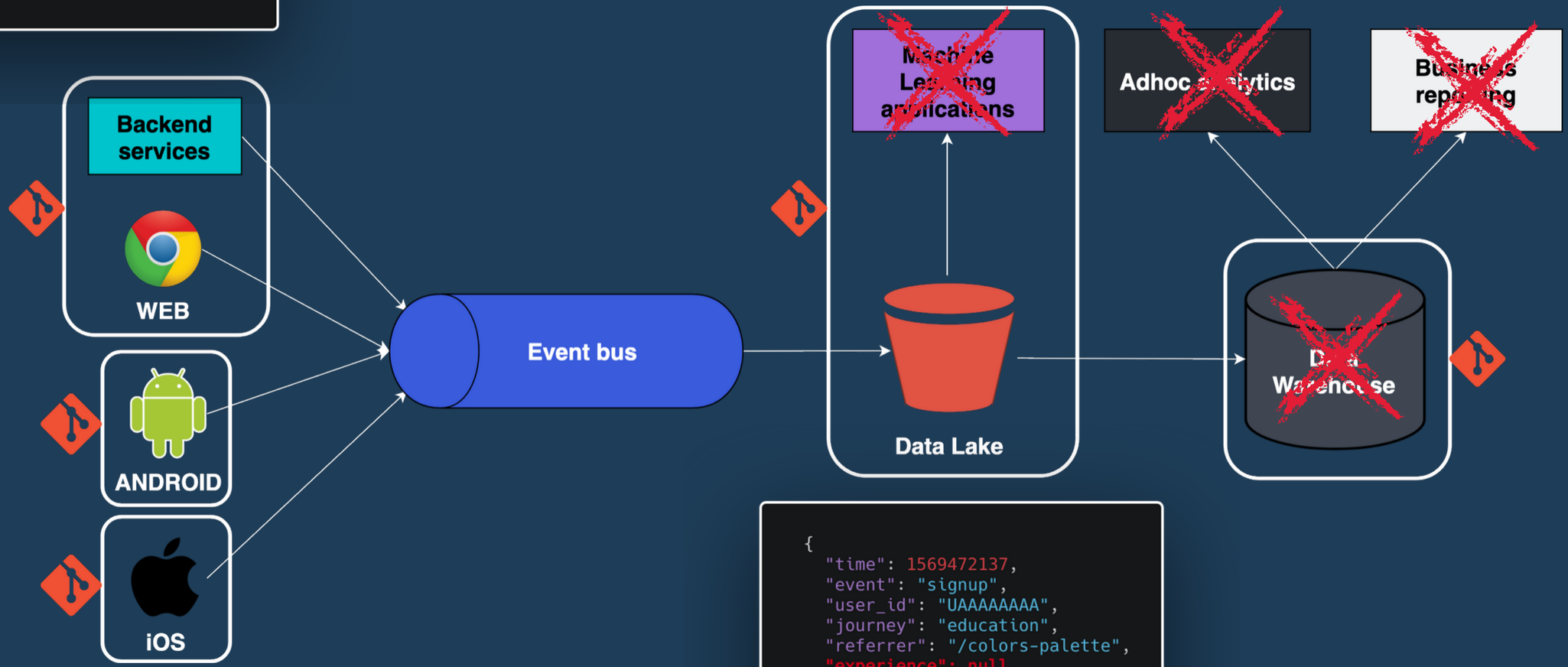


Git repository overlay



Example: Removing a field

```
{
  "time": 1569472137,
  "event": "signup",
  "user_id": "UAAAAAAAA",
  "journey": "education",
  "referrer": "/colors-palette",
- "experience": "ANDROID",
+ "platform": "ANDROID",
}
```




```
{
  "time": 1569472137,
  "event": "signup",
  "user_id": "UAAAAAAAA",
  "journey": "education",
  "referrer": "/colors-palette",
  "experience": null,
  "platform": "ANDROID",
}
```

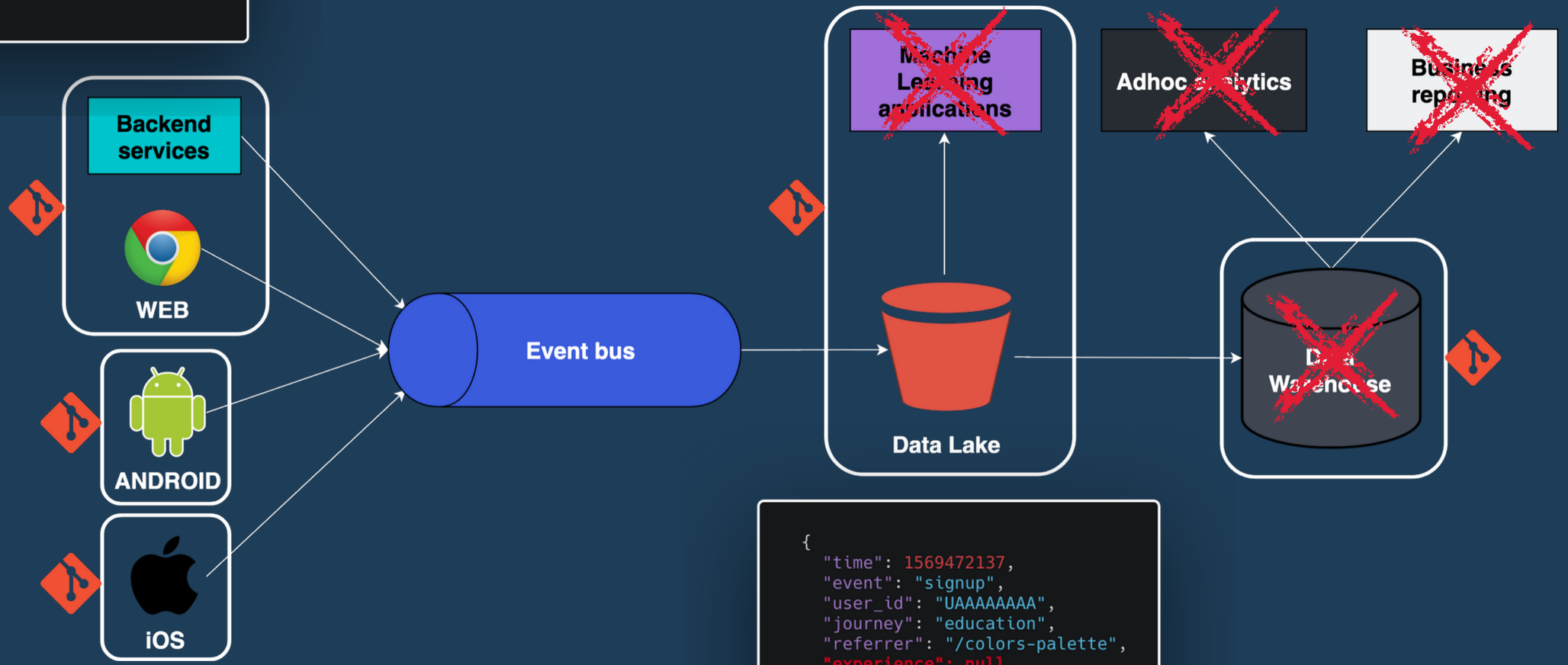
Problems

1. Multiple (conflicting 😖) sources of truth for schemas
2. Schema review and sharing
3. Backwards & forwards compatibility

Issue 1 - Multiple sources of truth

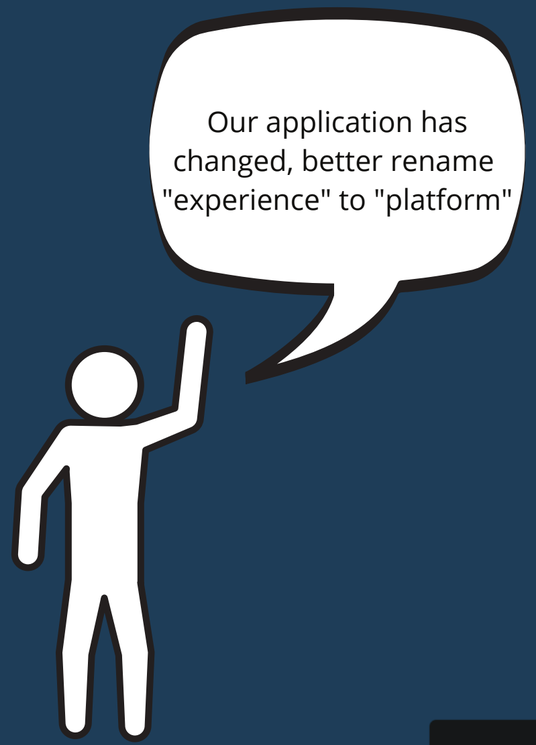
Example: Removing a field

```
{
  "time": 1569472137,
  "event": "signup",
  "user_id": "UAAAAAAAA",
  "journey": "education",
  "referrer": "/colors-palette",
- "experience": "ANDROID",
+ "platform": "ANDROID",
}
```




```
{
  "time": 1569472137,
  "event": "signup",
  "user_id": "UAAAAAAAA",
  "journey": "education",
  "referrer": "/colors-palette",
  "experience": null,
  "platform": "ANDROID",
}
```

Producer/Consumer



Producer

Repo X/Team A

```
Signup signup = Signup.builder()
  .withTime(1569472137)
  .withEvent("signup")
  .withUserId("UAAAAAAAA")
  .withJourney("education")
  .withReferrer("/colors/palette")
 -.withExperience("ANDROID")
 +.withPlatform("ANDROID")
  .build();
output = new EventStream();
signup.writeTo(output);
```

Producer code (Java)

```
{
  "time": 1569472137,
  "event": "signup",
  "user_id": "UAAAAAAAA",
  "journey": "education",
  "referrer": "/colors-palette",
- "experience": "ANDROID",
+ "platform": "ANDROID",
}
```

JSON payload

Consumer

Repo Y/Team B

```
@dataclass
class Signup:
     time: int
     event: str
     user_id: str
     journey: str
     referrer: str
     experience: str

input = EventStream()
signup.deserialize(input.read())
```

Consumer code (Python)

Consequences

```
{
  "time": 1569472137,
  "event": "signup",
  "user_id": "UAAAAAAAA",
  "journey": "education",
  "referrer": "/colors-palette",
- "experience": "ANDROID",
+ "platform": "ANDROID",
}
```

JSON payload

```
SELECT COUNT(DISTINCT(user_id))
    FROM warehouse.signup
    GROUP BY experience;
```

SQL

```
val signupDF = spark.read
    .parquet("/event/signup")
    .as[SignupEvent]
signupDF
    .groupBy("experience")
    .agg(countDistinct("user_id"))
```

Spark

(Silent) Failures

Q: How do we keep production and consumption of events in sync?

A: Schemas

Enters Protobuf

```
syntax = "proto2";

message Signup {
    required int64 time = 1;
    required string event = 2;
    required string user_id = 3;
    required string journey = 4;
    optional string referrer = 5;
    required string experience = 6;
}
```

Language/platform-neutral extensible mechanism for serialising structured data

Interface definition language (IDL)

```
syntax = "proto2";

message Signup {
    required int64 time = 1;
    required string event = 2;
    required string user_id = 3;
    required string journey = 4;
    optional string referrer = 5;
    required string experience = 6;
}
```

```
Signup signup = Signup.builder()
  .withTime(1569472137)
  .withEvent("signup")
  .withUserId("UAAAAAAAA")
  .withJourney("education")
  .withReferrer("/colors/palette")
  .withExperience("ANDROID")
  .build();
```

```
@dataclass
class Signup:
    time: int
    event: str
    user_id: str
    journey: str
    referrer: str
    experience: str
```

Single schema definition

Multiple code representation

Serialisation (binary) format

Producer

Repo X

```java
Signup signup = Signup.builder()
  .withTime(1569472137)
  .withEvent("signup")
  .withUserId("UAAAAAAAA")
  .withJourney("education")
  .withReferrer("/colors/palette")
  .withExperience("ANDROID")
  .build();
ouput = new EventStream()
signup.writeTo(output)
```

Producer code (Java)

Consumer

```python
@dataclass
class Signup:
     time: int
     event: str
     user_id: str
     journey: str
     referrer: str
     experience: str

input = EventStream()
signup.deserialize(input.read())
```

Consumer code (Python)

Repo Y

Vanilla protobuf

- Java
- Python
- Go
- Dart
- C++
- C#
- Binary SerDe

Canva's requirements

- Java
- Swift
- Kotlin
- Scala
- JS/Typescript
- Json/Jackson library SerDe

Q: How do you generate custom code from proto files?

A: Build a tool or reuse an existing one

Enters Datumgen

```
syntax = "proto2";

message Signup {
    required int64 time = 1;
    required string event = 2;
    required string user_id = 3;
    required string journey = 4;
    optional string referrer = 5;
    required string experience = 6;
}
```

```
class CaseClassModel {
  public String getName();

  public List<CaseClassField> getFields();
}

class CaseClassField {
  public Boolean isRequired();

  public String getJsonKey();

  public String getName();

  public String getType();
}
```

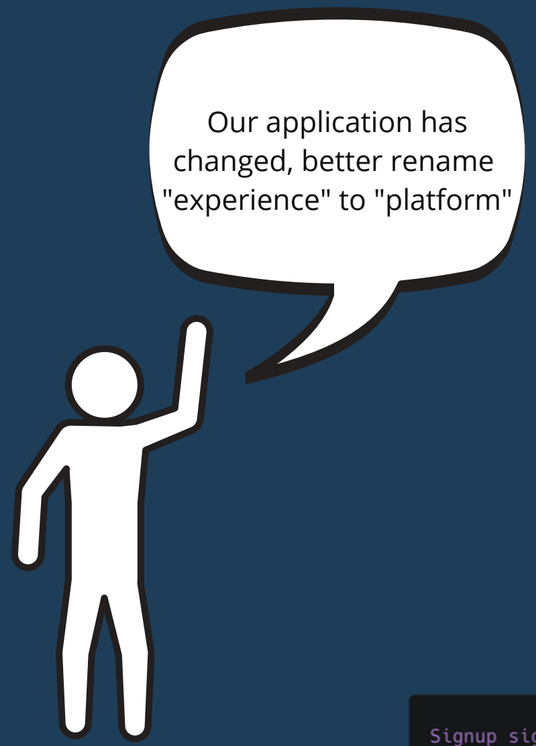
```
caseClass(caseClass) ::= <<
<if(caseClass.fields)>
case class <caseClass.name>(
  <caseClass.fields:caseClassParamDeclaration(); separator=",\n">
)<endif>
>>

caseClassParamDeclaration(field) ::= <%
<if(field.required)>
@JsonProperty(value = "<field.jsonKey>", required = true)<\ >
<else>
@JsonProperty("<field.jsonKey>")<\ >
<endif>
<field.name>:<\ >
<if(field.required)>
<field.type>
<else>
Option[<field.type>]
<endif>
%>
```

StringTemplate

Rendering

```
case class Signup(
  @JsonProperty("time") time: Long,
  @JsonProperty("event") event: String,
  @JsonProperty("user_id") userId: String,
  @JsonProperty("journey") journey: String,
  @JsonProperty("referrer") referrer: Option[String],
  @JsonProperty("experience") experience: String,
)
```

Improved producer/consumer



Producer

Repo X/Team A

```
Signup signup = Signup.builder()
  .withTime(1569472137)
  .withEvent("signup")
  .withUserId("UAAAAAAAA")
  .withJourney("education")
  .withReferrer("/colors/palette")
 -.withExperience("ANDROID")
 +.withPlatform("ANDROID")
  .build();
output = new EventStream();
signup.writeTo(output);
```

```
{
  "time": 1569472137,
  "event": "signup",
  "user_id": "UAAAAAAAA",
  "journey": "education",
  "referrer": "/colors-palette",
- "experience": "ANDROID",
+ "platform": "ANDROID",
}
```

JSON payload

```
syntax = "proto2";

message Signup {
    required int64 time = 1;
    required string event = 2;
    required string user_id = 3;
    required string journey = 4;
    optional string referrer = 5;
   -required string experience = 6;
   +optional string platform = 7;
}
```

Repo X/Team A

```
@dataclass
class Signup:
     time: int
     event: str
     user_id: str
     journey: str
     referrer: str
    -experience: str
    +platform: str

input = EventStream()
Signup.deserialize(input.read())
```

Repo Y/Team B

Consumer

Summary - Issue 1

-Multiple definitions of the same structure

+Single canonical schema definition

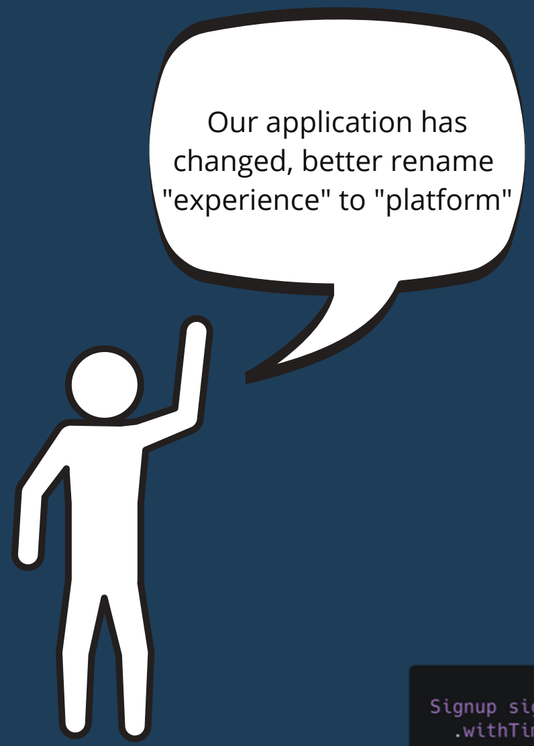
-Handwritten code in 2+ languages

+Automated code generation for any language

Issue 2 - Schema review and code sharing

Having a single schema definition actually doesn't keep people from making breaking changes

Driven by consumers or SME



Producer

Consumer

```
{
  "time": 1569472137,
  "event": "signup",
  "user_id": "UAAAAAAAA",
  "journey": "education",
  "referrer": "/colors-palette",
- "experience": "ANDROID",
+ "platform": "ANDROID",
}
```

JSON payload

```
Signup signup = Signup.builder()
  .withTime(1569472137)
  .withEvent("signup")
  .withUserId("UAAAAAAAA")
  .withJourney("education")
  .withReferrer("/colors/palette")
 -.withExperience("ANDROID")
 +.withPlatform("ANDROID")
  .build();
output = new EventStream();
signup.writeTo(output);
```

```
syntax = "proto2";

message Signup {
    required int64 time = 1;
    required string event = 2;
    required string user_id = 3;
    required string journey = 4;
    optional string referrer = 5;
   -required string experience = 6;
   +optional string platform = 7;
}
```

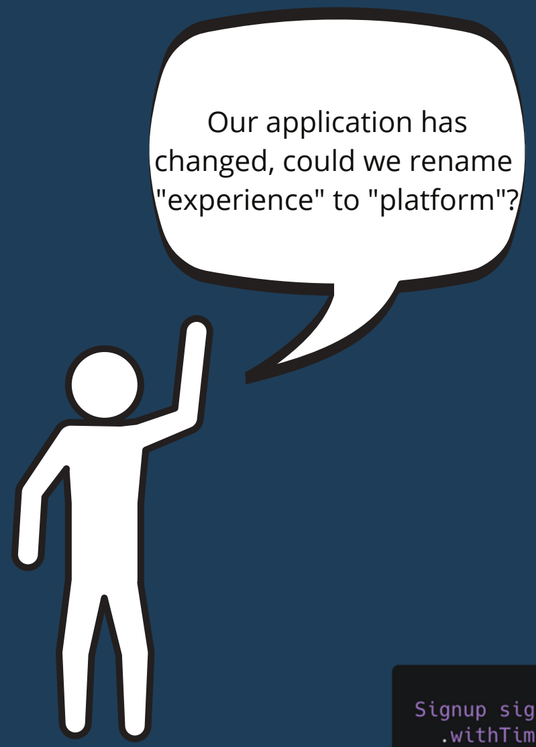
```
@dataclass
class Signup:
     time: int
     event: str
     user_id: str
     journey: str
     referrer: str
    -experience: str
    +platform: str

input = EventStream()
Signup.deserialize(input.read())
```

Repo X/Team A

Repo Y/Team B

Repo X/Team A → Repo Y/Team B

Review process (denied)



Producer

Consumer

```
{
  "time": 1569472137,
  "event": "signup",
  "user_id": "UAAAAAAAA",
  "journey": "education",
  "referrer": "/colors-palette",
- "experience": "ANDROID",
+ "platform": "ANDROID",
}
```

JSON payload

```
Signup signup = Signup.builder()
  .withTime(1569472137)
  .withEvent("signup")
  .withUserId("UAAAAAAAA")
  .withJourney("education")
  .withReferrer("/colors/palette")
  .withExperience("ANDROID")
  .build();
ouput = new EventStream()
signup.writeTo(output)
```

```
syntax = "proto2";

message Signup {
    required int64 time = 1;
    required string event = 2;
    required string user_id = 3;
    required string journey = 4;
    optional string referrer = 5;
   -required string experience = 6;
   +optional string platform = 7;
}
```

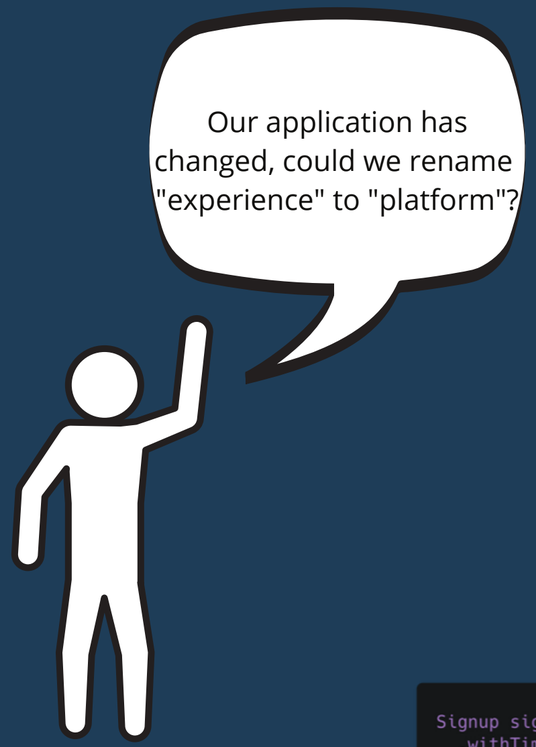
```
@dataclass
class Signup:
     time: int
     event: str
     user_id: str
     journey: str
     referrer: str
     experience: str

input = EventStream()
signup.deserialize(input.read())
```

Repo X/Team A

Repo Y/Team B

Repo Y/Team B

Review process (approved)



Producer

```
{
 "time": 1569472137,
 "event":"signup",
 "user_id": "UAAAAAAAA",
 "journey":"education",
 "referrer":"/colors-palette",
 "experience":"ANDROID",
+"brand":"canva"
}
```

JSON payload

Consumer

```
Signup signup = Signup.builder()
  .withTime(1569472137)
  .withEvent("signup")
  .withUserId("UAAAAAAAA")
  .withJourney("education")
  .withReferrer("/colors/palette")
  .withExperience("ANDROID")
  +.withBrand("canva")
  .build();
output = new EventStream();
signup.writeTo(output);
```

Repo X/Team A

```
syntax = "proto2";

message Signup {
    required int64 time = 1;
    required string event = 2;
    required string user_id = 3;
    required string journey = 4;
    optional string referrer = 5;
    required string experience = 6;
   +optional string brand = 7;
}
```

Repo Y/Team B

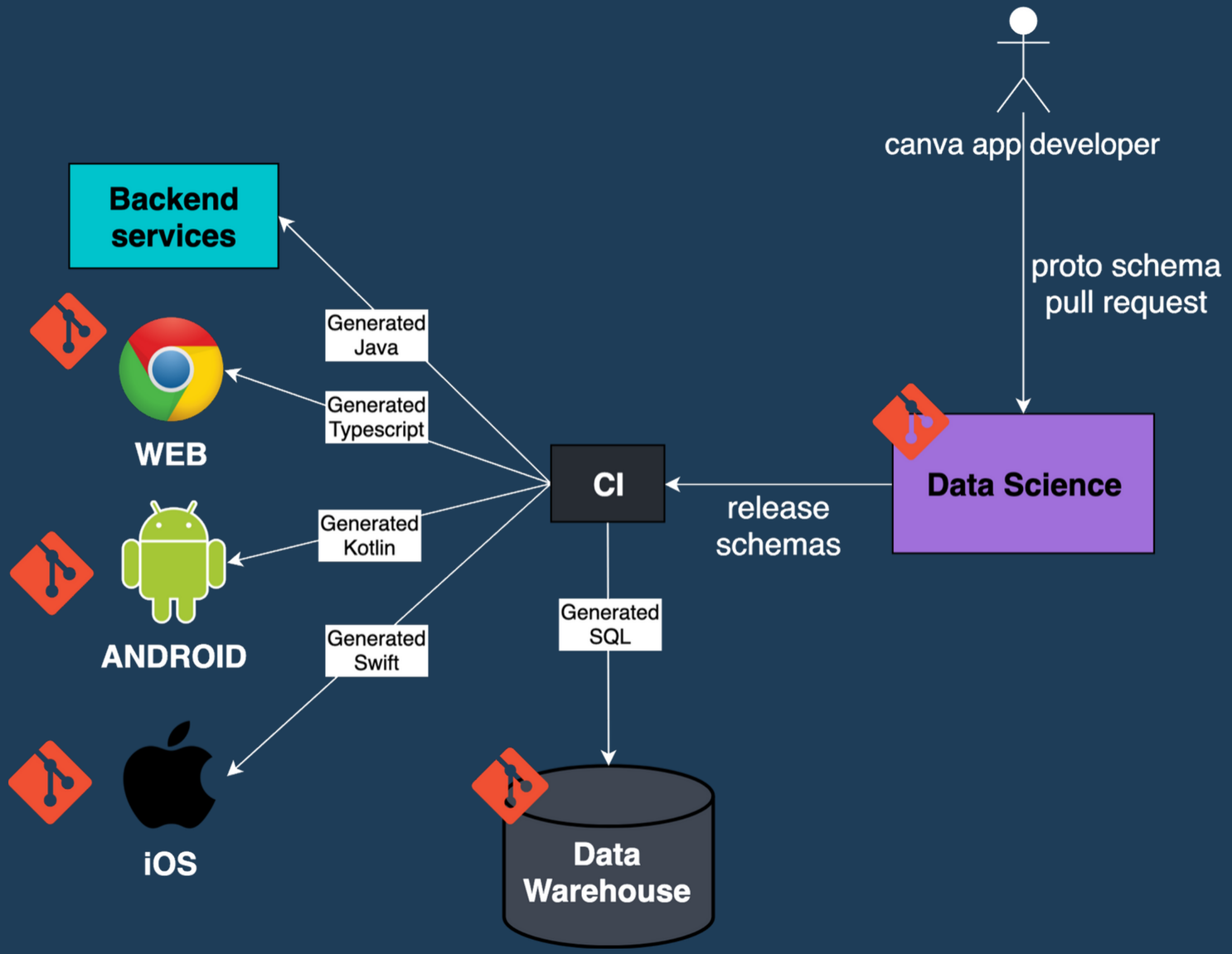
```
@dataclass
class Signup:
     time: int
     event: str
     user_id: str
     journey: str
     referrer: str
     experience: str
     +brand: str

input = EventStream()
Signup.deserialize(input.read())
```

Repo Y/Team B

Q: How do you propagate generated code changes to all its dependencies

A: CI

Automated schema sharing



Summary - Issue 2

-Unforeseen/bad schema evolution

+Consumer ownership of schemas, peer review process, guardians of compatibility

-Generated code out of sync

+Automated code propagation

Issue 3 - Backwards and forwards compatibility

Maintaing full transitive compatibilty

Compatibility types

Compatibility Type	Changes allowed	Check against which schemas	Upgrade first
`BACKWARD`	• Delete fields • Add optional fields	Last version	Consumers
`BACKWARD_TRANSITIVE`	• Delete fields • Add optional fields	All previous versions	Consumers
`FORWARD`	• Add fields • Delete optional fields	Last version	Producers
`FORWARD_TRANSITIVE`	• Add fields • Delete optional fields	All previous versions	Producers
`FULL`	• Modify optional fields	Last version	Any order
`FULL_TRANSITIVE`	• Modify optional fields	All previous versions	Any order
`NONE`	• All changes are accepted	Compatibility checking disabled	Depends

https://docs.confluent.io/current/schema-registry/avro.html#compatibility-types

BACKWARD COMPATIBILITY

New consumers need to be able to read both new messages and the previous message version.

Makes sense for:

- RPC services
- Regular release schedule
- Monorepos

Compatibility Type	Changes allowed	Check against which schemas	Upgrade first
`BACKWARD`	• Delete fields • Add optional fields	Last version	Consumers
`BACKWARD_TRANSITIVE`	• Delete fields • Add optional fields	All previous versions	Consumers
`FORWARD`	• Add fields • Delete optional fields	Last version	Producers
`FORWARD_TRANSITIVE`	• Add fields • Delete optional fields	All previous versions	Producers
`FULL`	• Modify optional fields	Last version	Any order
`FULL_TRANSITIVE`	• Modify optional fields	All previous versions	Any order
`NONE`	• All changes are accepted	Compatibility checking disabled	Depends

https://docs.confluent.io/current/schema-registry/avro.html#compatibility-types

FORWARD

New messages need to be able to be read by **old** consumers.

Compatibility Type	Changes allowed	Check against which schemas	Upgrade first
BACKWARD	• Delete fields • Add optional fields	Last version	Consumers
BACKWARD_TRANSITIVE	• Delete fields • Add optional fields	All previous versions	Consumers
FORWARD	• Add fields • Delete optional fields	Last version	Producers
FORWARD_TRANSITIVE	• Add fields • Delete optional fields	All previous versions	Producers
FULL	• Modify optional fields	Last version	Any order
FULL_TRANSITIVE	• Modify optional fields	All previous versions	Any order
NONE	• All changes are accepted	Compatibility checking disabled	Depends

https://docs.confluent.io/current/schema-registry/avro.html#compatibility-types

Makes sense for:

- Decoupled services reading from an event bus (eg. Kafka)
- Regular release schedule
- Monorepos

Full transitive compatibility

What:
Consumers need to be able to read:

- the current message version,
- all the previous versions,
- all future versions.

Why:
Need to be able to query and make comparisons against long term historical data

How:

- Only add optional fields, or remove unused optional fields
- **No enums!**

CAVEATS*

You can cheat these compatibility rules under 3 circumstances:

1. You backfill all old data into the new schema and migrate all downstream users

2. You are can tolerate data loss when a new producer or consumer is deployed

3. You can guarantee you have no downstream users (yet)

How do we maintain full transitive compatibility?

Protolock

github.com/nilslice/protolock

"Protocol Buffer companion tool. Track your .proto files and prevent changes to messages and services which impact API compatibility."

signup_event.proto

```
syntax = "proto2";

message Signup {
    required int64 time = 1;
    required string event = 2;
    required string user_id = 3;
    required string journey = 4;
    optional string referrer = 5;
    required string experience = 6;
}
```

> protolock init

proto.lock

```
{
  "definitions": [
    {
      "protopath": "signup_event.proto",
      "def": {
        "messages": [
          {
            "name": "Signup",
            "fields": [
              {
                "id": 1,
                "name": "time",
                "type": "int64"
              },
              {
                "id": 2,
                "name": "event",
                "type": "string"
              },
              ...
              ...
              ...
              {
                "id": 6,
                "name": "experience",
                "type": "string"
              }
            ]
          }
        ],
        "package": {
          "name": "com.canva.example"
        }
      }
    }
  ]
}
```

signup_event.proto

```
syntax = "proto2";

message Signup {
    required int64 time = 1;
    required string event = 2;
    required string user_id = 3;
    required string journey = 4;
    optional string referrer = 5;
   -required string experience = 6;
   +optional string platform = 7;
}
```

```
> protolock commit
```

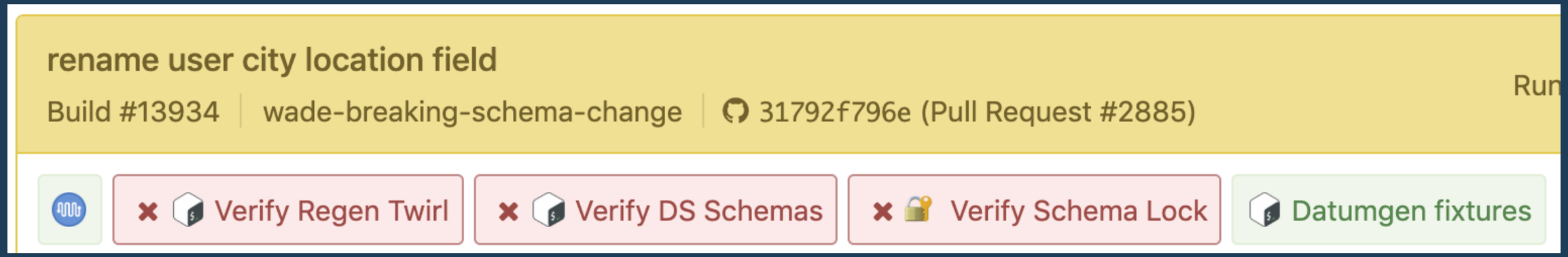
CONFLICT: "Signup" ID: "6" has been removed [signup_event.proto]

CONFLICT: "Signup" field: "experience" has been removed [signup_event.proto]

Protolock Compatibility Rules

- No changing field IDs
- No changing field types
- No changing field names
- No removing required fields
- No adding new required fields

Social engineering via red bubbles



Bonus benefits of generating code from schemas

- Generated producers and consumers
- Standardise events across apps / platforms:
  eg. web, backend, iOS, Android
- Abstract the event bus
- Abstract the storage layer
- Abstract monitoring
- Wide ranging refactors become trivial
- Better schema design when reviews are centralised
- Easy to assign data owners

Cons of generating code from schemas

- Very hard to retrofit into an existing system
- Requires buy-in from all data producers
- Schema sharing across multiple repos is eventually consistent
- Big tooling investment

Summary

Before:

- Multiple manually written sources of truth
- No best practice around event design
- Breaking changes everywhere

Summary

Before:

- Multiple manually written sources of truth
- No best practice around event design
- Breaking changes everywhere

After:

- Generated producer and consumer clients from proto
- Centralised schema review process and publishing mechanism
- Automated CI checks enforce full transitive compatibility

We're hiring

- Data Engineers
- Machine Learning Engineers
- Data Analysts

https://www.canva.com/careers/jobs/

Questions