Large scale data capture and experimentation platform at Grab

September 2018

Product Insights & Experimentation at Grab

Experimentation Platform aims to accelerate innovation, by allowing us to find and focus on key ideas evaluated through rigorous regime of online controlled experiments. It also promotes the culture of data driven decision making through experimentation.

Scribe is our in house mobile and backend event collection solution. It helps us track, monitor and analyse any mobile event such as user clicks, booking rides as well as mobile logs, data usage and more.

Team

Experimentation Platform

Feature Toggles

Grab's Product Insights & Experimentation platform provides a dynamic **feature toggle** capability to our engineering, data, product and even business teams. Feature toggles also let teams modify system behavior **without changing code**.

At Grab, we use feature toggles to:

1. Gate **feature deployment** in production for keeping new features hidden until product and marketing teams are ready to share.
2. Run **experiments (A/B tests)** by dynamically changing feature toggles for specific users, rides, etc. For example, a feature can only appear to a particular group of people while experiment is running (treatment group).

More on Feature Toggles: https://martinfowler.com/articles/feature-toggles.html

Our Mission

The Grab Experimentation Platform aims to accelerate innovation, by allowing us to find and focus on key ideas evaluated through rigorous regime of online controlled experiments. It also promotes the culture of data driven decision making through experimentation.

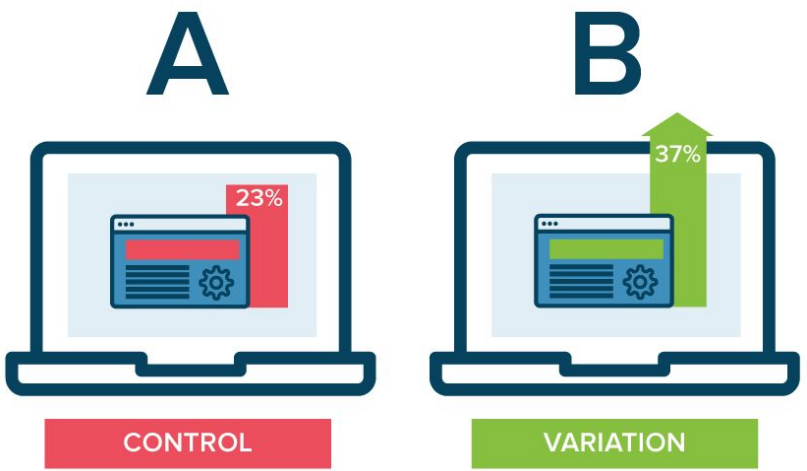
To get there, we must lower the cost of experimentation significantly.

What is Experimentation or Testing

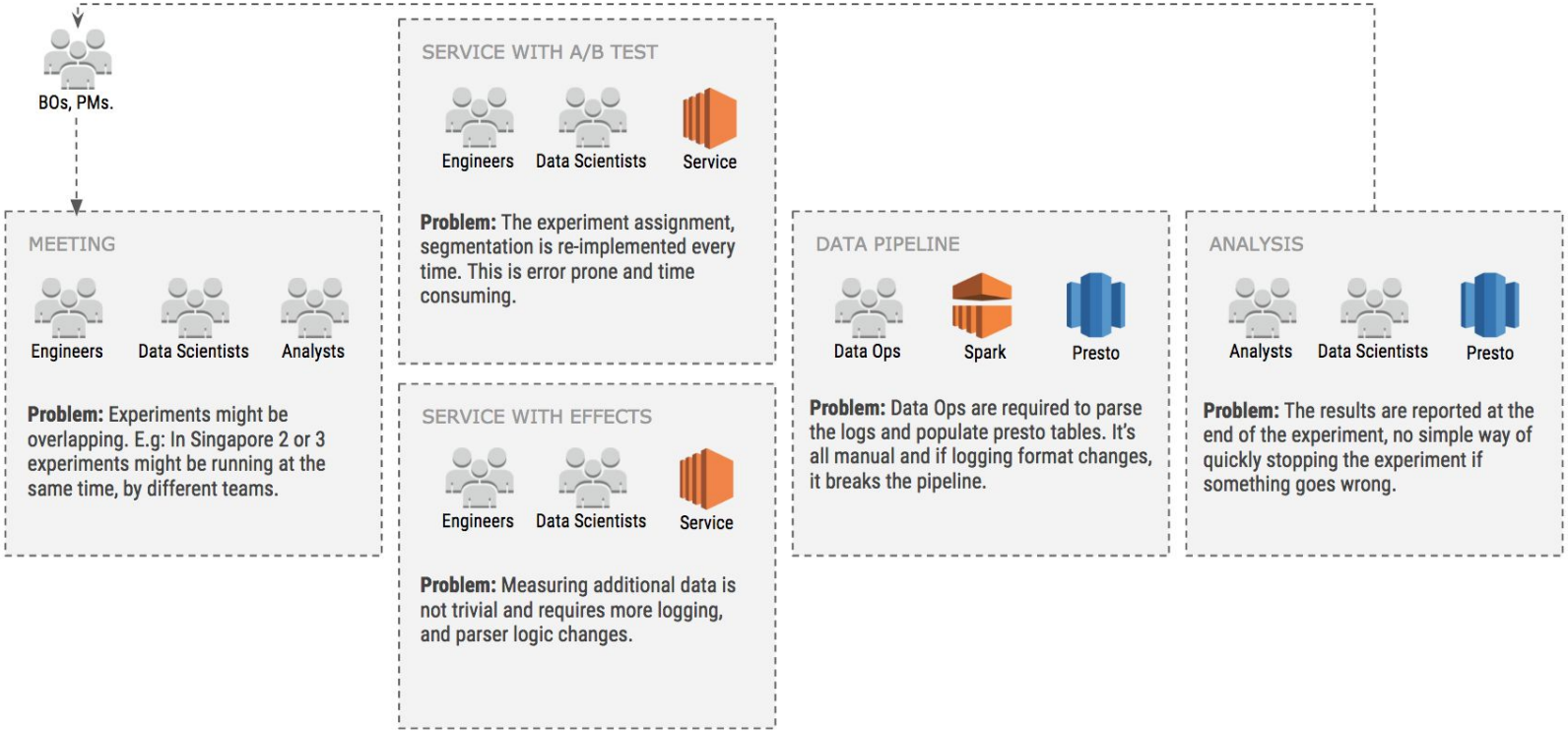
Testing is a method of **comparing two or more versions** of a system against each other to determine which one **performs best**.

For example, in an experiment where two or more variants of a page are shown to users at random, and statistical analysis is used to determine which variation performs better for a given conversion goal.

Well known types of testing: **A/B**, **Multivariate**, **Factorial** etc.



Long Time Ago… Prior to Experimentation Platform



Example:
Automated Message Experiment

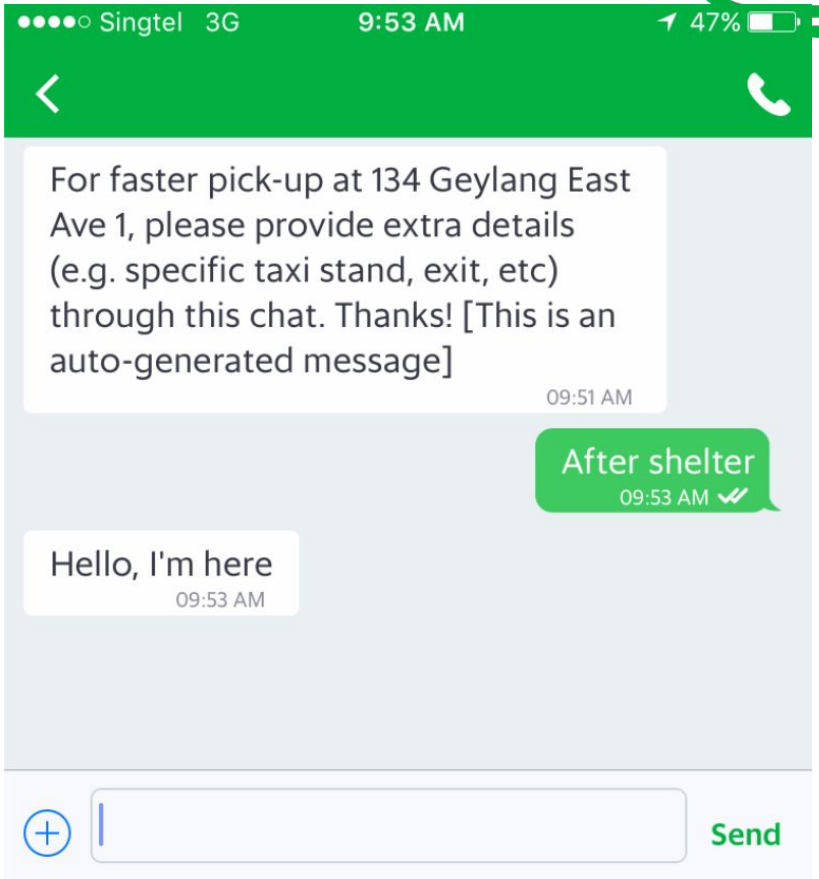
Automated Message Experiment

Problem: We wanted to determine the **best delay** to send an automated message to passengers asking for extra details.

- **70%** of bookings had no message
- **10%** of bookings got a message after **30s**
- **10%** of bookings got a message after **60s**
- **10%** of bookings got a message after **90s**

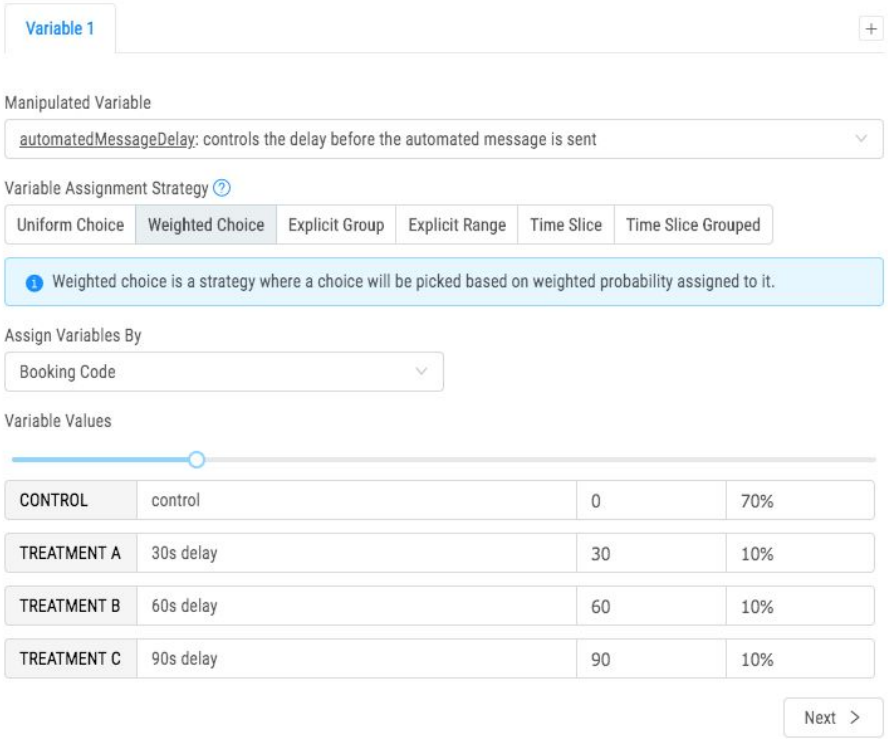
The experiment ran on **Singapore GC, GC+ and JustGrab** for a period of one week.

The **hypothesis** was that sending a message sooner will **consistently reduce cancellations**.

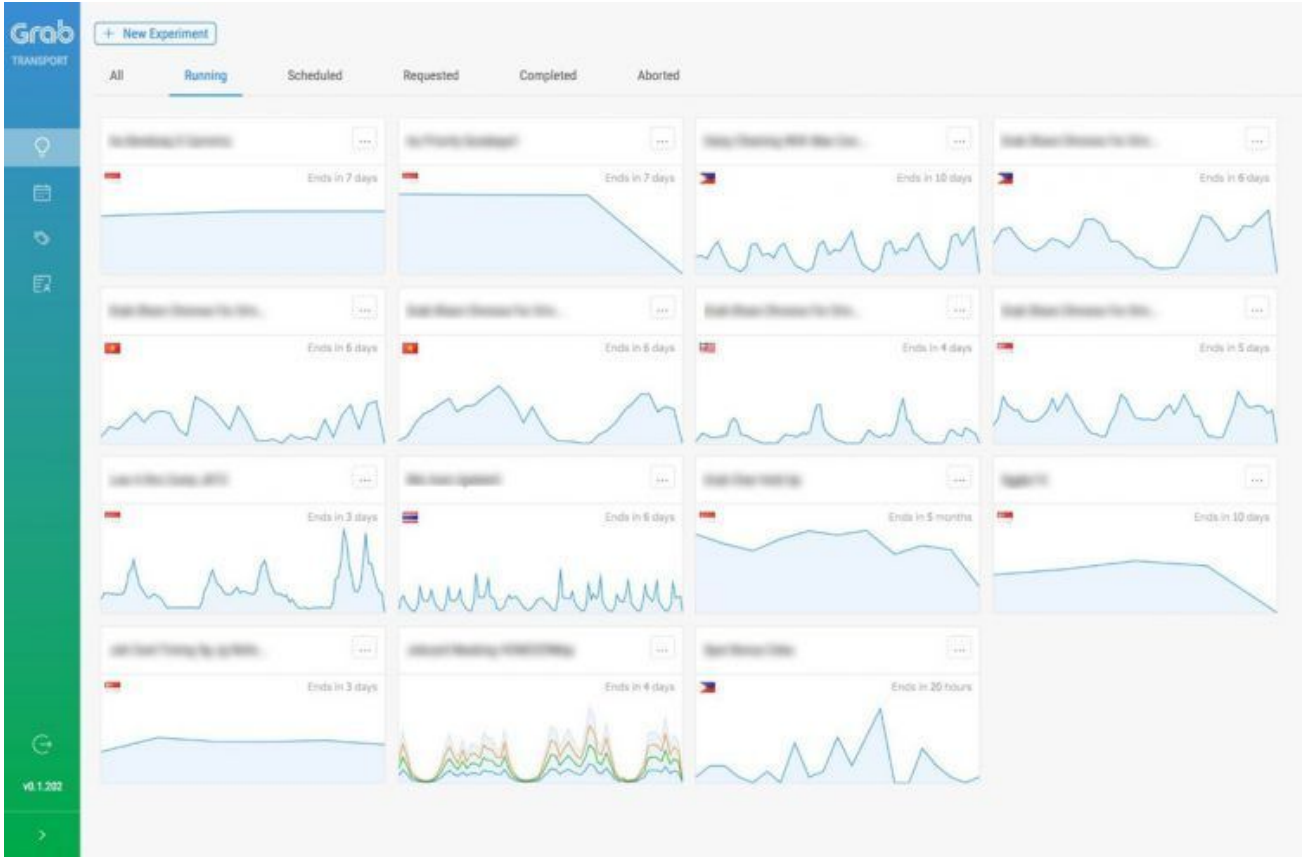


Experiment Definition

```
"domain": "primary",
"name": "primary.automatedChatMessage",
"variables": [{
      "name": "automatedMessageDelay",
      "strategy": "weightedChoice",
      "facets": ["bkg"],
      "choices": [
         { "value": 0, "weight": 0.70 },
         { "value": 30, "weight": 0.10 },
         { "value": 60, "weight": 0.10 },
         { "value": 90, "weight": 0.10 }]
}],
"constraints": [
   {"target": "ts", "op": ">", "value": "1400113600"},
   {"target": "ts", "op": "<", "value": "1400718400"},
   {"target": "svc", "op": "in", "value": "[1,2,3]"}
]
```



Experimentation Platform Portal



Engineering: Using the SDK

In Grab Chat Service

SDK is used to retrieve a value for the variable based on the configuration. Default value will be used if no experiment is running or constraints do not match.

```
facets := sdk.NewFacets().
      City(city).
      Vehicle(vehicleType).
      Passenger(passenger).
      Booking(booking)

delay := X.GetVariable(ctx,"automatedMessageDelay",
      facets).Int64(0)

sendMessage(delay) // logic
```

In Booking Service

SDK is used to instrument required cancellation metrics.

```
facets := sdk.NewFacets().
      City(req.CityID).
      Vehicle(vehicleType).
      Passenger(passenger).
      Booking(booking)

X.Track(ctx,"booking.cancelled", reason, facets)
```

Feature Toggles

How we did Feature Toggles before...

Our legacy feature toggling system was essentially a library shipped with all of our Golang services that wrapped calls to a shared Redis. Retrieving values involved network calls and local caching to support our scale, but slowly it started to become a single point of failure for the company as the number of backend microservices grew.

```
// Retrieve a feature flag using our legacy system
flag := sitevar.GetFeatureFlag("myFeature", 10, false)
```

Golang SDK

Our Go SDK provides a very simple client which will **take care of everything**.

```
type Client interface {
	// GetVariable with name of the variable
	GetVariable(ctx context.Context, name string, facets *Facets) Variable

	// Track an event with a value and metadata
	Track(ctx context.Context, eventName string, value float64, facets *Facets)

	// InjectLocally injects the facets to the local golang context. This will only be
	// applied to this context and will not be propagated throughout the service stack.
	InjectLocally(ctx context.Context, values Facets) context.Context

	// InjectGlobally injects the facets to the global tracing baggage. This allows you
	// to propagate facet values across services.
	InjectGlobally(span opentracing.Span, values Facets)
}
```

Events and Facets

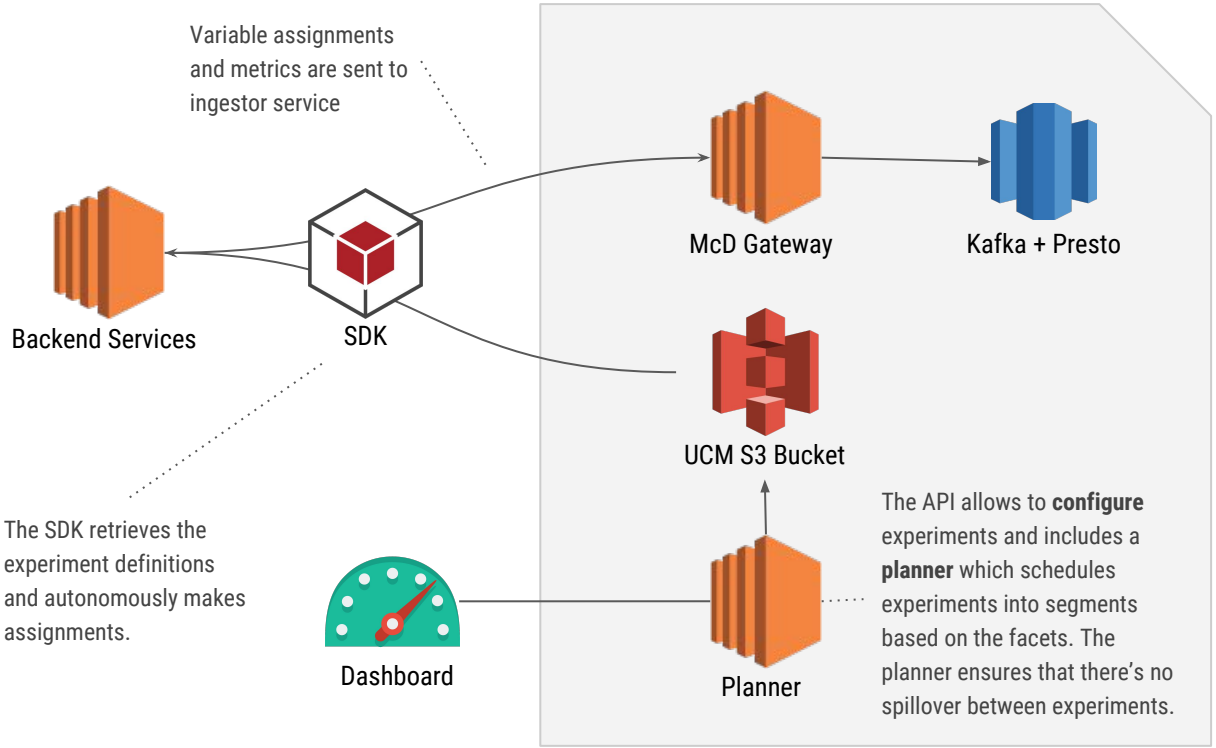
The facets will need to be filled as **much as possible**.

```
Facets {
    Passenger  int64  // The passenger identifier
    Driver     int64  // The driver identifier
    Service    int64  // The equivalent to a vehicle type
    Booking    string // The booking code
    Location   string // The location (geohash or coordinates)
    Session    string // The session identifier
    Request    string // The request identifier
    Device     string // The device identifier
    Tag        string // The user-defined tag
    ...
}
```

The facets are set using a 3-step override (ie.: implicit global, implicit local, explicit).

- **Step 1 - Global Implicit.** If some facets are found in open tracing baggage span, set those facet values.
- **Step 2 - Local Implicit.** If some facets are found in local golang context, set and override those facet values.
- **Step 3 - Explicit.** The facets provided explicitly in the GetVariable/Track call should override all implicit ones.

Architectural Overview



Reliability

We designed the platform to be as reliable as possible, making sure that we do not affect our customers (you guys) in any way if we're down.

Our data is stored in UCM S3 bucket, and if McD is down the data simply won't be written but none of backend services will be affected.

Tracking Events with Scribe

McD - Data is Lovin’ It

Problem statement

To enable the construction of platforms that will provide us **valuable business insight** and improve the **quality of our products and services**.

What we built

McD - a general-purpose event ingestion system to capture useful mobile and backend events such as impressions, user interactions (eg. clicks), performance metrics (eg. battery life).

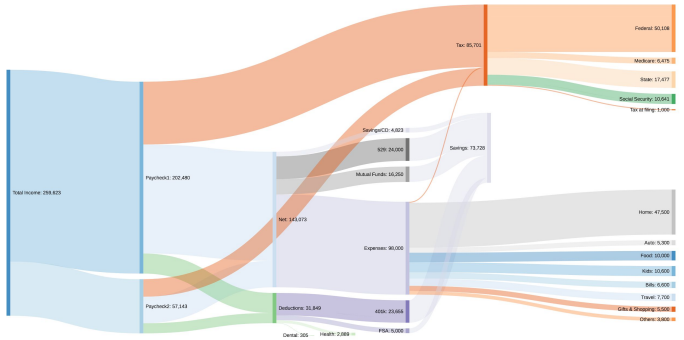
Funnels and Experimentation

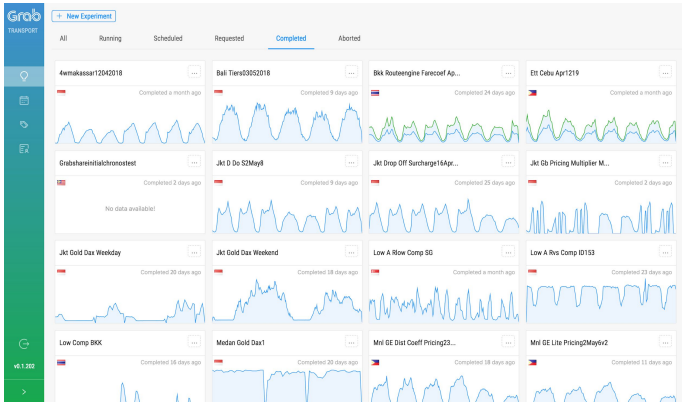
Funnels

Need to capture unified funnels (backend and mobile), especially with proliferation of Grab Apps.

Experimentation

Need of fully-fledged and unified experimentation platform for our mobile and backend.





Logging, Monitoring and Profiles

Monitoring and Logging
Need of simple monitoring and debuggability tool for our mobile apps (passenger, driver, food…)

User Profiles
Need to capture comprehensive click streams and user profiles.

SDKs for every platform

- Available for Golang, Android and iOS
- Efficient, batched, binary event transmission
- Dynamic configuration of the SDK (per device)
- Built-in persistence & QoS to avoid data loss
- Automatic tracking of common events (e.g.: “app start”)
- Integrated feature flags & A/B testing
- Dart and JavaScript SDKs are planned

Reliably Track Anything

```
facets := sdk.NewFacets().
     Booking(req.BookingCode).
     Custom(“key1”, “some value”).
     Custom(“key2”, “some other value”)

X.Track(ctx,"customEvent", 42, facets)
```

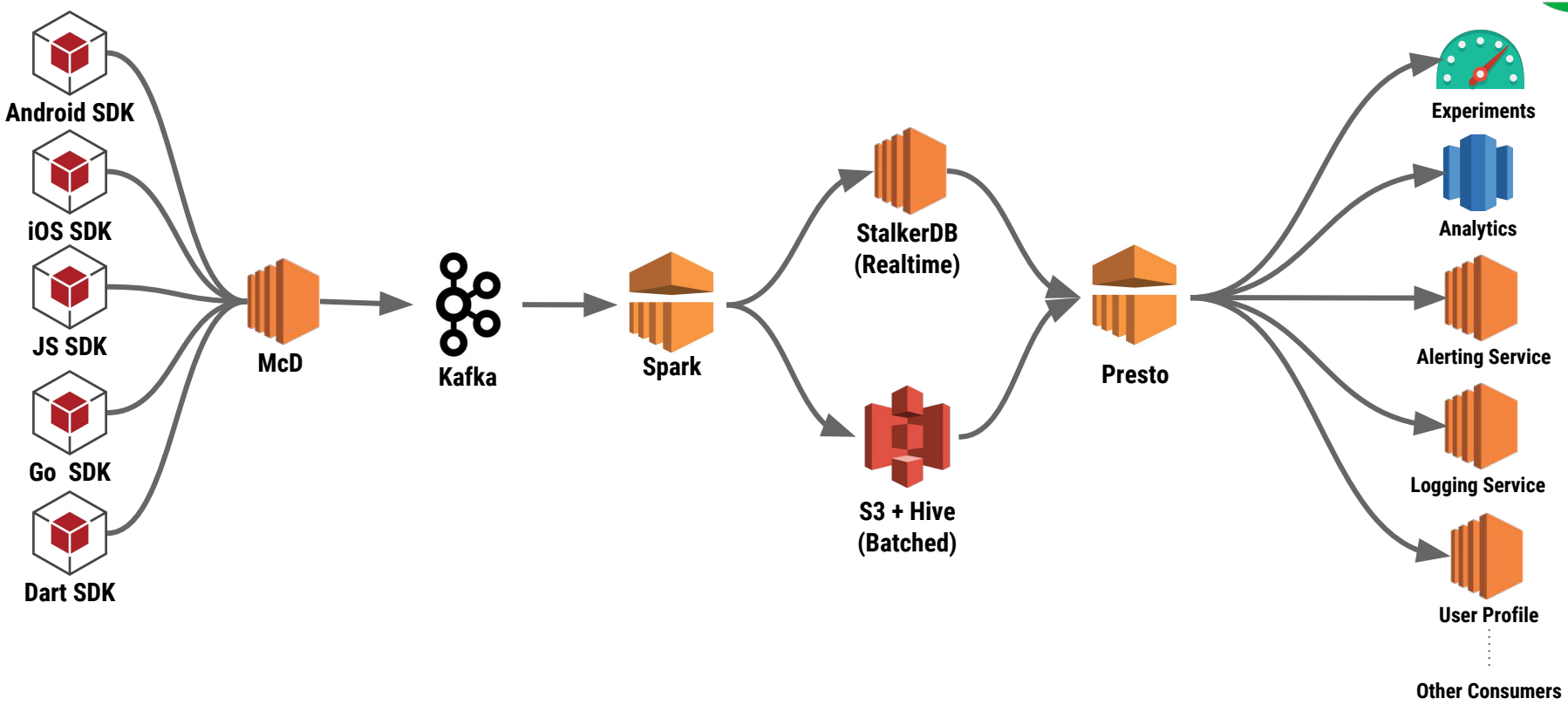
```
SELECT avg(val) mean, city
FROM eventlog
WHERE event = 'myservice.customEvent'
GROUP BY city
ORDER BY 1 DESC
LIMIT 100
```

Our SDK can be used to track any event.

- The **name of the event**, which gets prefixed by the service name.
- The numeric **value of the event**.
- The **facets** representing contextual information about this event. Users are encouraged to provide as much information as possible, for example, passenger ID, booking code, driver ID.

Scaling our Data Pipeline

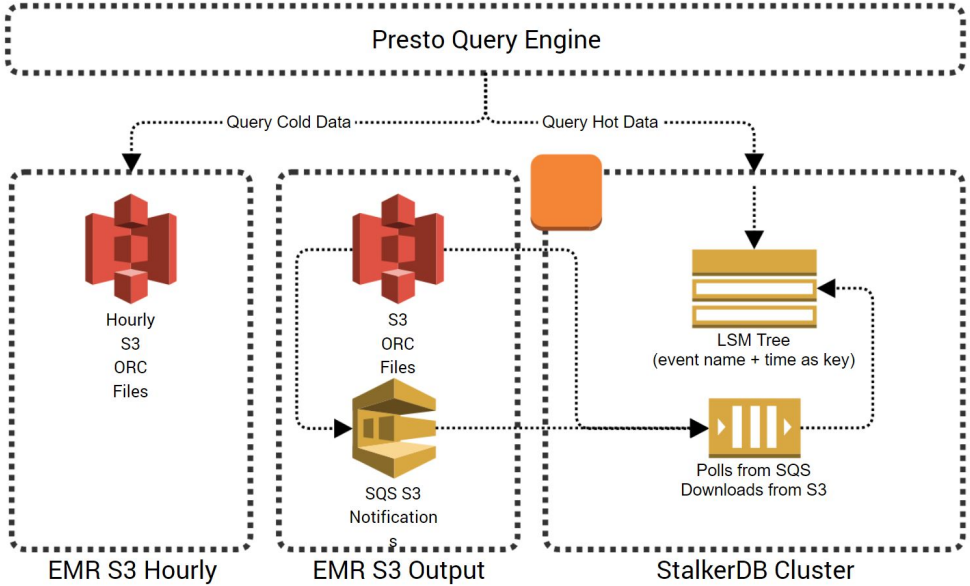
Data Collection Architecture



Querying Terabytes in Milliseconds

Simple and Reliable

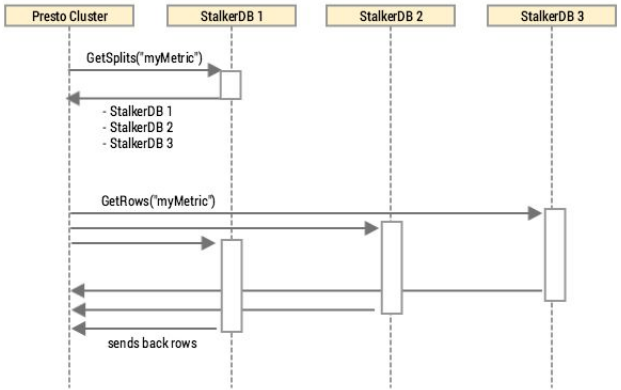
One of the goals of StalkerDB is simplicity. We achieve that by making sure that system itself is **not responsible for things like data transformation and re-partitioning** of data but simply **ingests** and **serves** data to Presto, nothing else.



How Ingestion Happens

```
// Append appends a set of events to the storage. It needs ingestion time and an event name to create
// a key which will then be used for retrieval.
func (s *Server) Append(payload []byte) error {
    schema, err := orc.SplitByColumn(payload, column, func(event string, columnChunk []byte) bool {
        _, err := orc.SplitBySize(columnChunk, 25000, func(chunk []byte) bool {

            // Create a new block to store from orc buffer
            blk, _ := block.FromOrc(chunk)

            // Encode the block
            buffer, _ := blk.Encode()

            // Append this block to the store
            s.store.Append( newKey(event, time.Unix(0, tsi)), buffer, time.Hour)
            return false
        })
        return err != nil
    })
    return nil
}
```

Presto Interaction



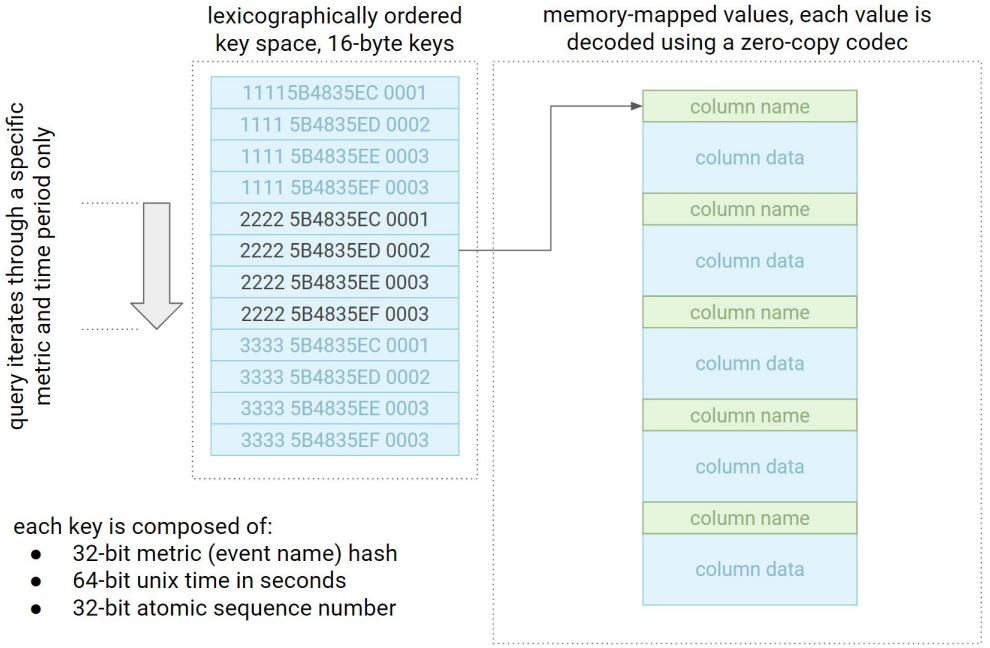

```
// PrestoGetSplits returns a batch of splits.
func (s *Server) PrestoGetSplits(...) {

    // Create a new query and validate it
    ...

    // We need to generate as many splits as we have nodes in our cluster. Each
    // split needs to contain the IP address of the node containing that split,
    // so Presto can reach it and request the data.
    batch = new(presto.PrestoThriftSplitBatch)
    for _, m := range s.cluster.Members() {
        for _, q := range queries {
            batch.Splits = append(batch.Splits, &presto.PrestoThriftSplit{
                SplitId: q.Encode(),
                Hosts:   []*presto.PrestoThriftHostAddress{{Host: m, Port: s.port}},
            })
        }
    }
    return
}
```

Combining LSMT with Columnar Zero-Copy Codec

TBs in Milliseconds

The keys are lexicographically ordered and when a query comes, it essentially seeks to the first key for that metric and stops iterating when either next metric is found or time bound is reached.

Decoding is efficient, without any unnecessary memory copy.



Thank you!