Threading Needles in a Haystack: Sessionizing Uber Trips in Real Time

Amey Chaugule
Uber Technologies, Inc.

Agenda

- The Marketplace Org @ Uber
- Sessionization use cases at Uber
- Some unique challenges about Sessions @ Uber
- Anatomy of an Uber Session
- Our Sessions DSL
- Sessions in Production
- Scale
- Q & A

The Uber Marketplace

Dynamic Pricing

We develop the systems, algorithms, and pricing structures to balance and optimize the marketplace in real time.

Intelligent Dispatch

We build and optimize the dispatching algorithms to efficiently match riders and drivers, lowering wait times and prices.

Driver Positioning

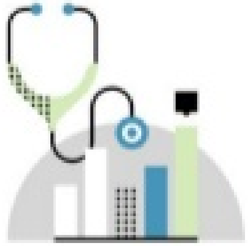
We provide guidance to drivers in real time to locations with the highest potential of earnings, using data and algorithmic approaches.

Marketplace Health

We build realtime diagnostic visual tools to identify opportunities for improving the experience in every neighborhood.

Realtime Forecasting

We analyze several data feeds to accurately predict city-specific traffic, routes, and demand patterns by time and location.

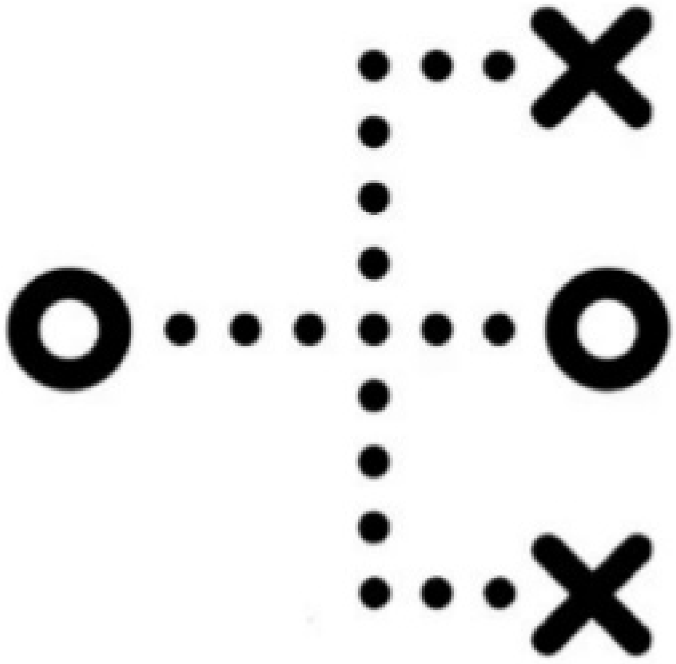


Marketplace Platform

We design, build, and operate the always-on platform for Uber to ensure that every Uber trip is a magical one.

The Uber Marketplace

In our **marketplace,** there are **models that describe the world** and the **decision engines** that act upon them.



Marketplace @ Uber

Real-time events in the **physical world** drive **marketplace dynamics** which then affect the **algorithmic engines** in the Marketplace which in turn influence the events in the real world in a **continuous feedback loop**.

Some examples of marketplace dynamics:

Supply
Demand
Forecast
Trips

Need for a sessionized view of the Uber experience

Given the scale and complexity of our systems, events are distributed across **multiple disparate real-time streams** over the twin dimensions of **time and space**.

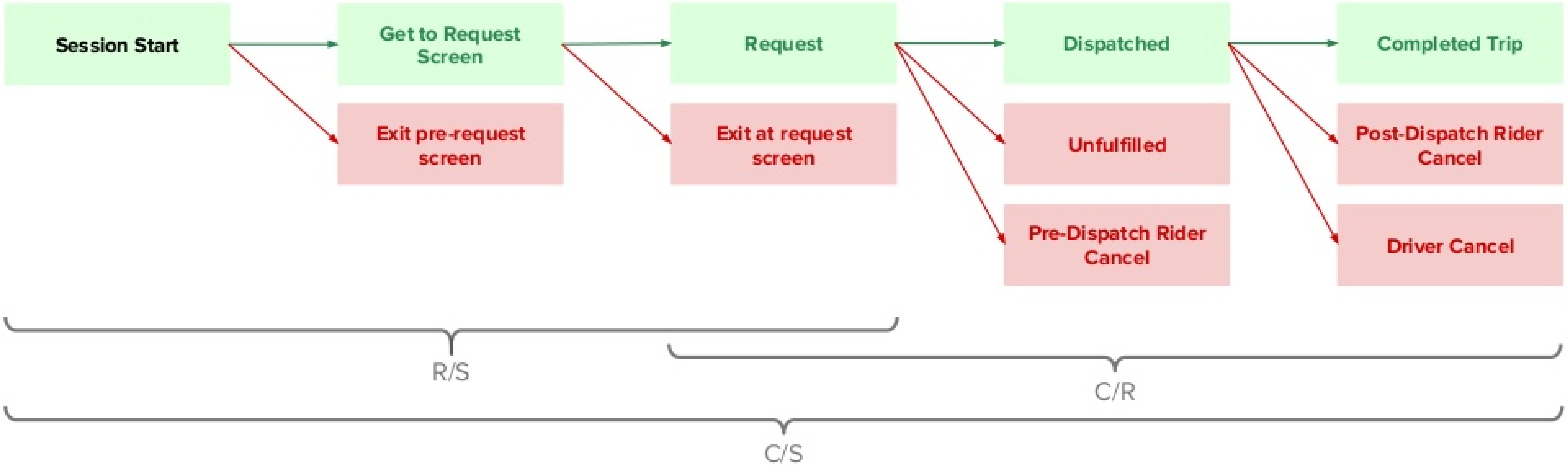
How do we **contextualize** these **event streams** so they can be **logically grouped together** and quickly surface useful information to **downstream decision engines**?

Real-time Use cases

For instance, the some algorithms need to adjust to **spikes in demand** to **balance the supply** in near real-time.

While some **machine learning-models** need information about **pre-request activities** in real-time.

To understand rider behaviour we need to understand the full user experience from start to finish.



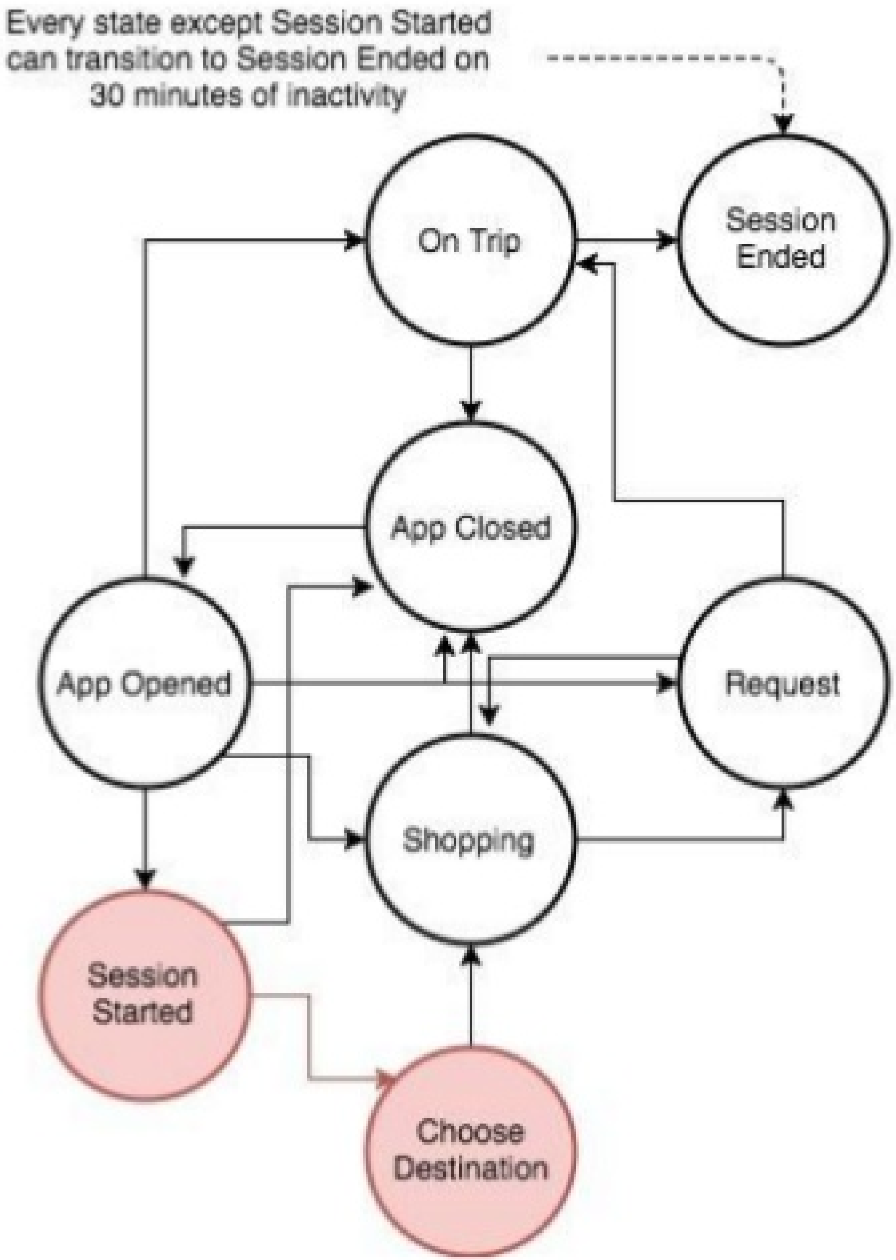
The definition of this experience, a **SESSION**, is critical to understanding our internal business operations.

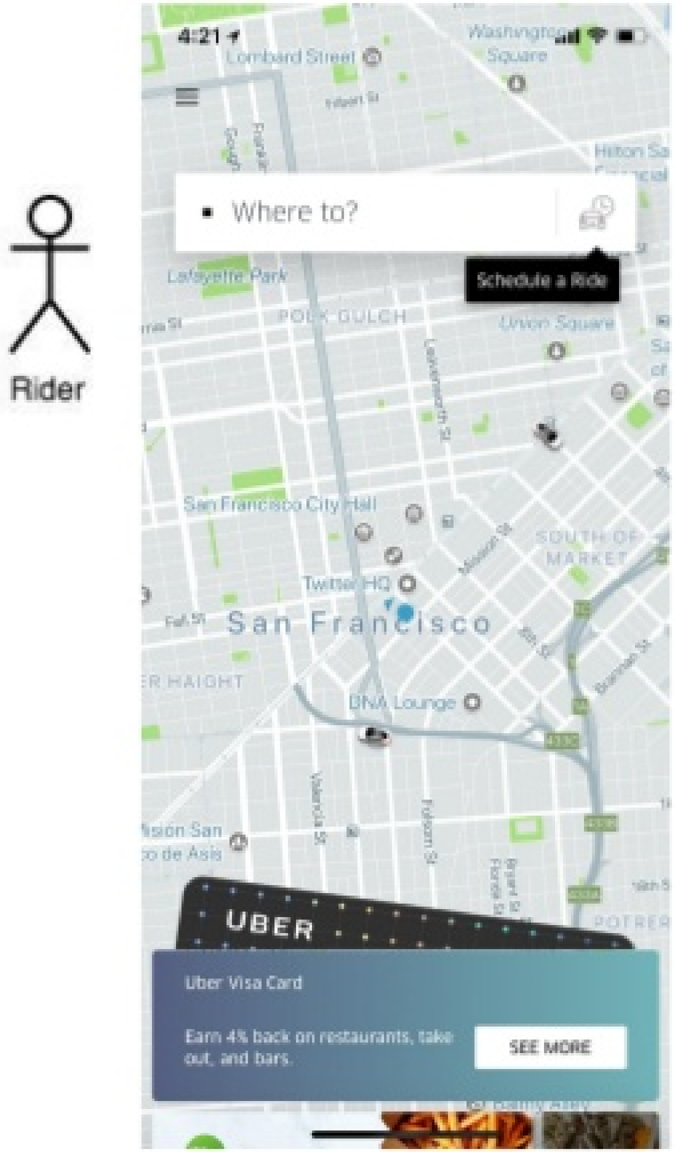
Some unique challenges

Given the ride-sharing marketplace Uber operates, our Sessions state machine needs to model interactions between **riders**, **driver partners** and **back-end systems**

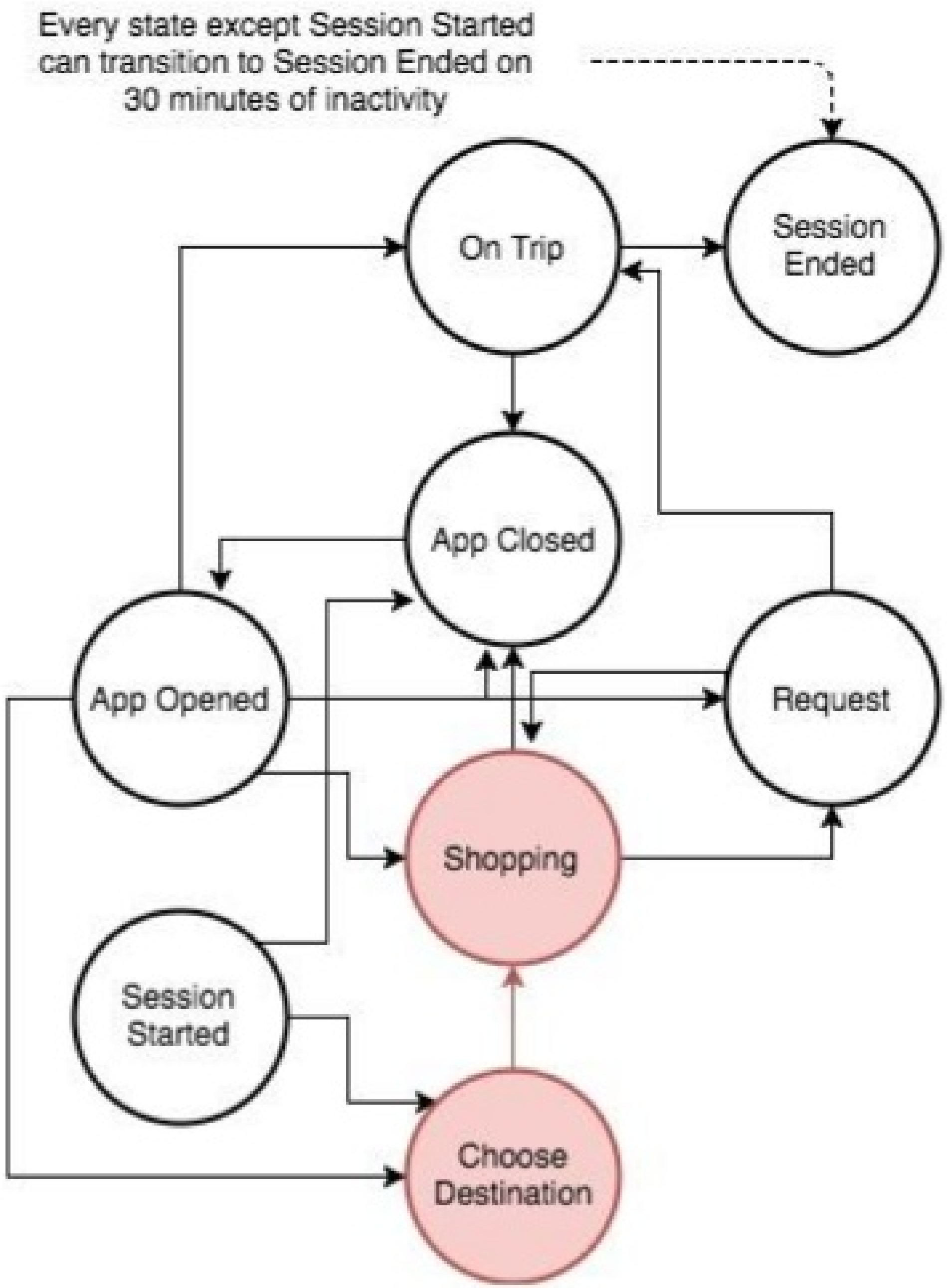
The **Anatomy** of an **Uber Session**

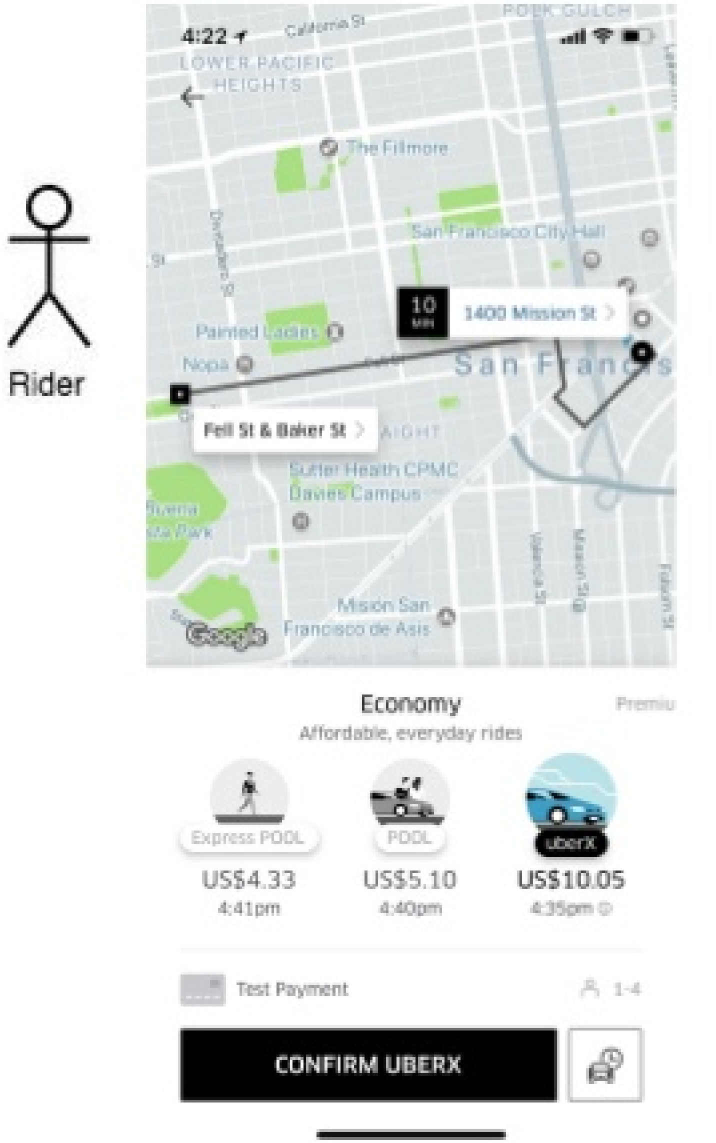
The Sessions State Machine



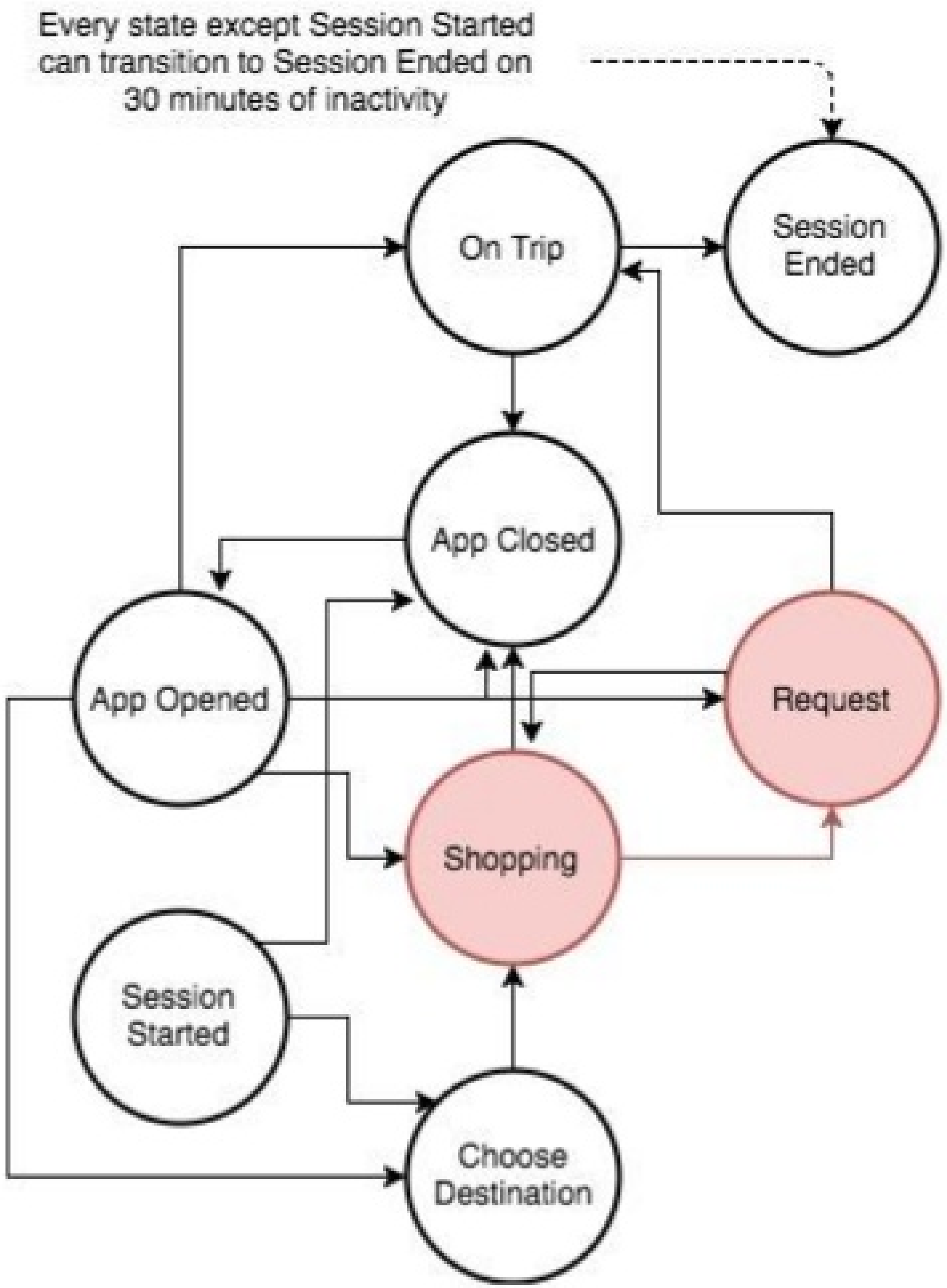


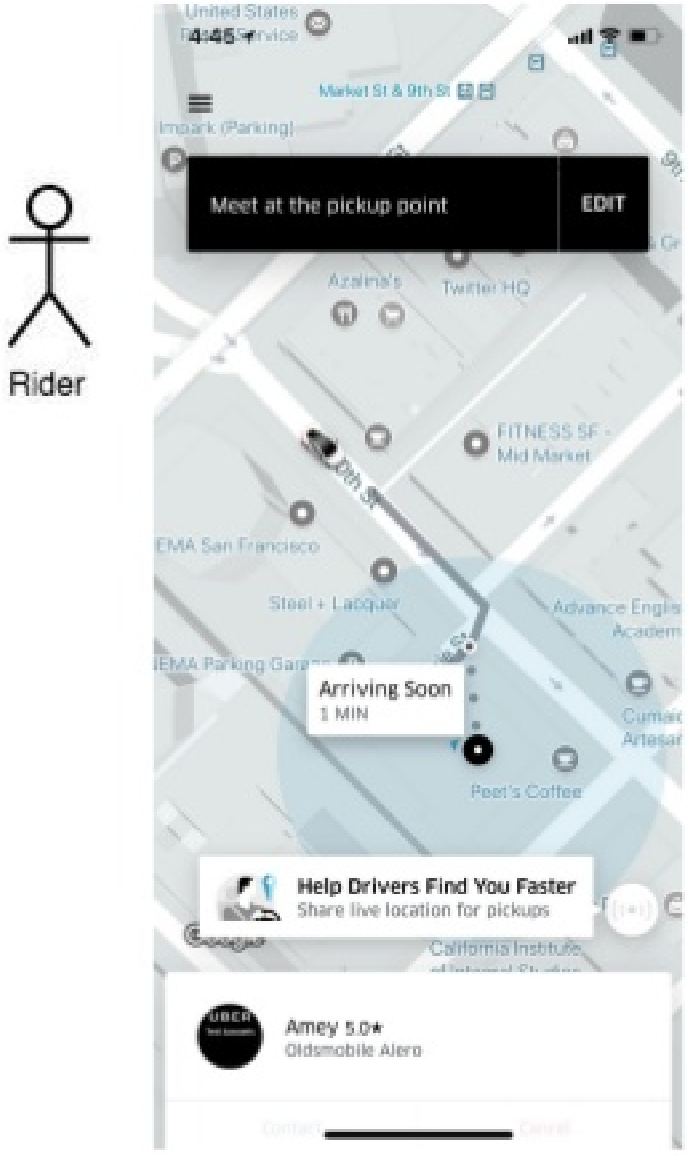
The Sessions State Machine - The shopping state



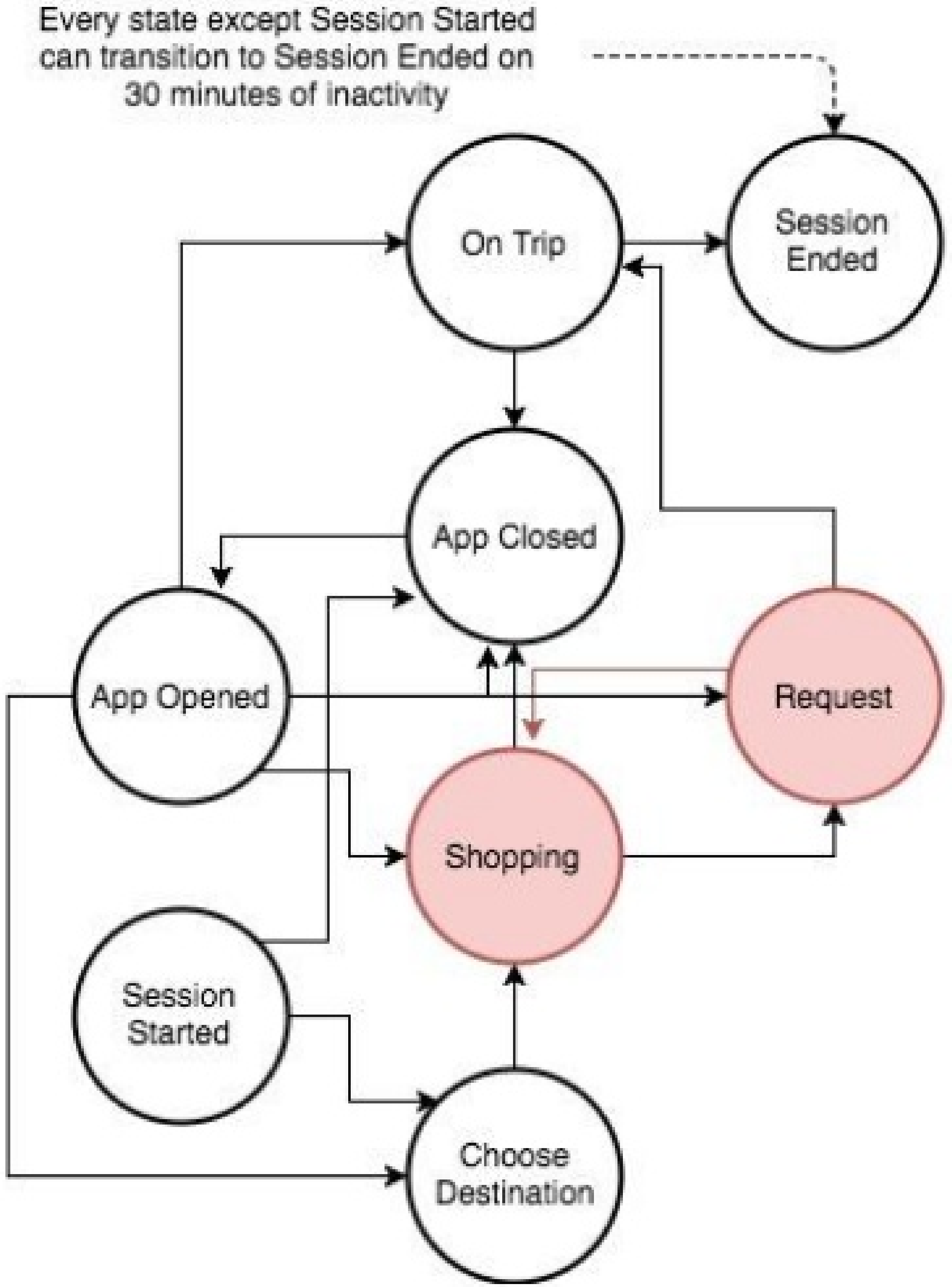


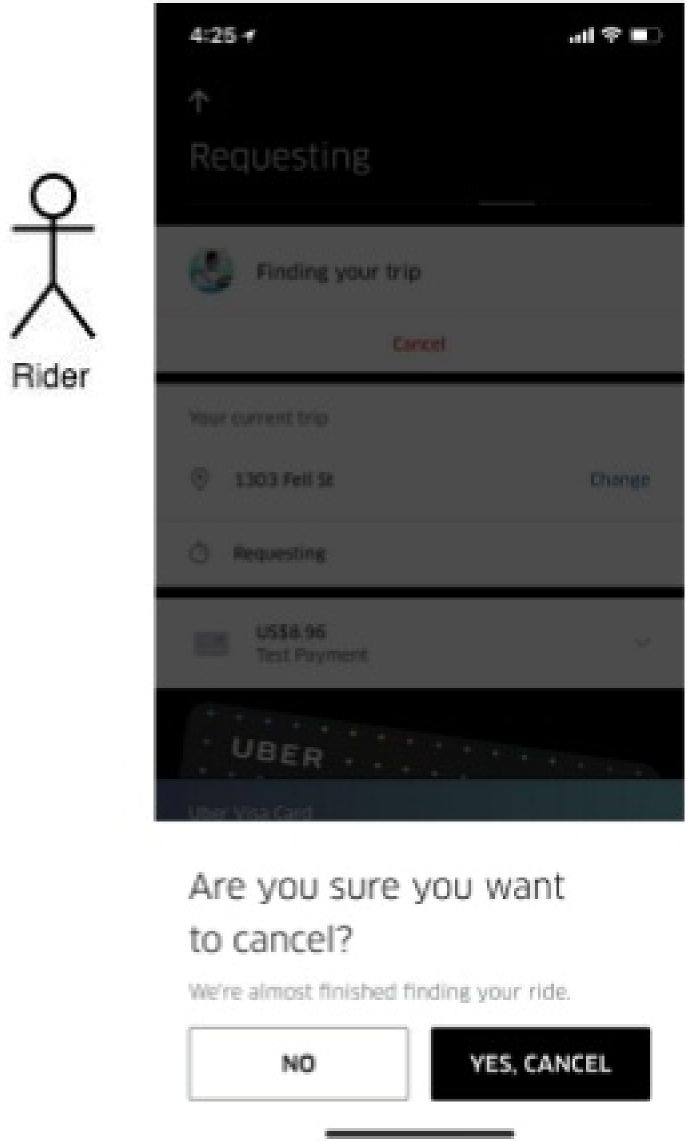
The Sessions State Machine - Requesting a ride



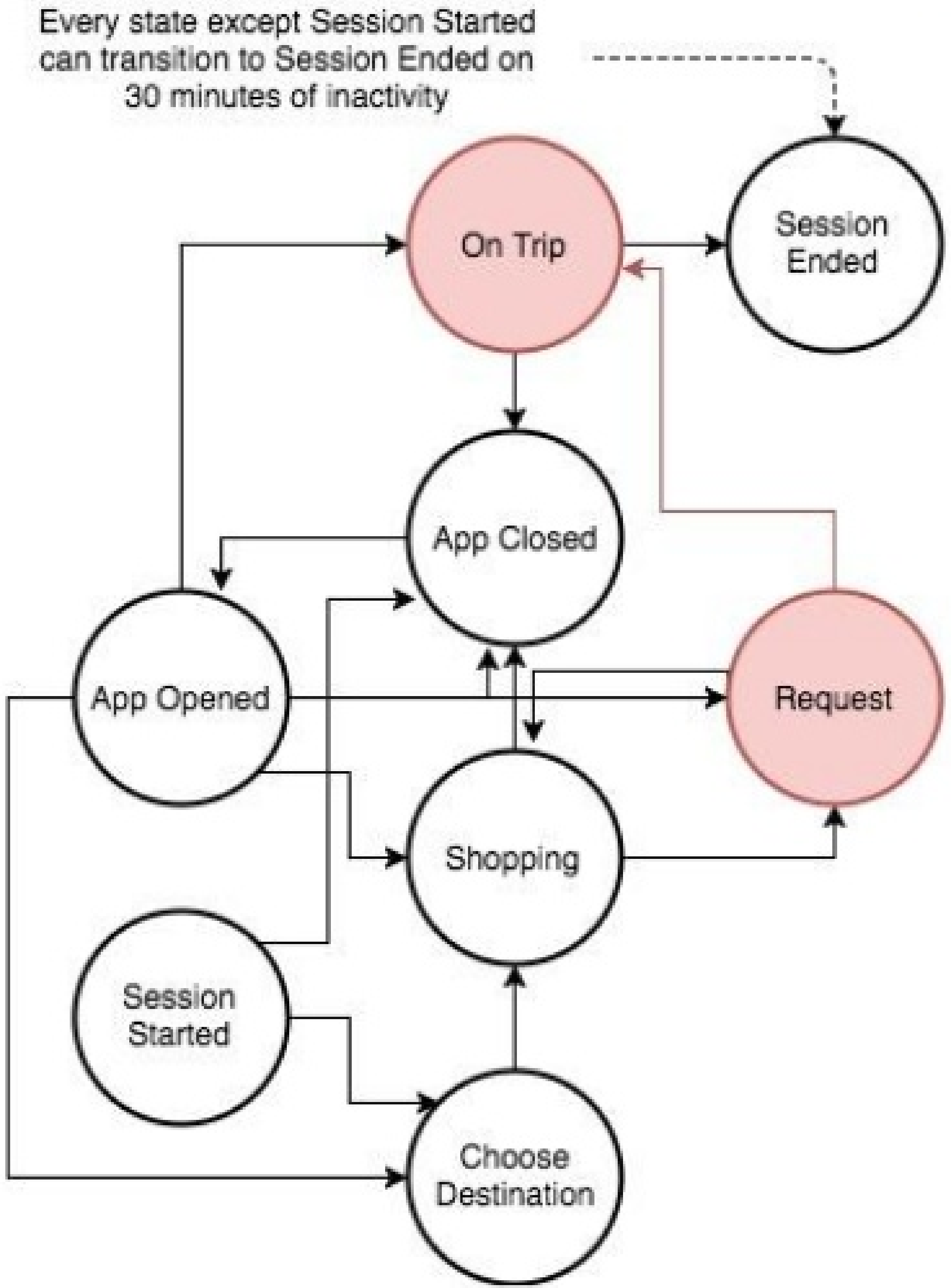


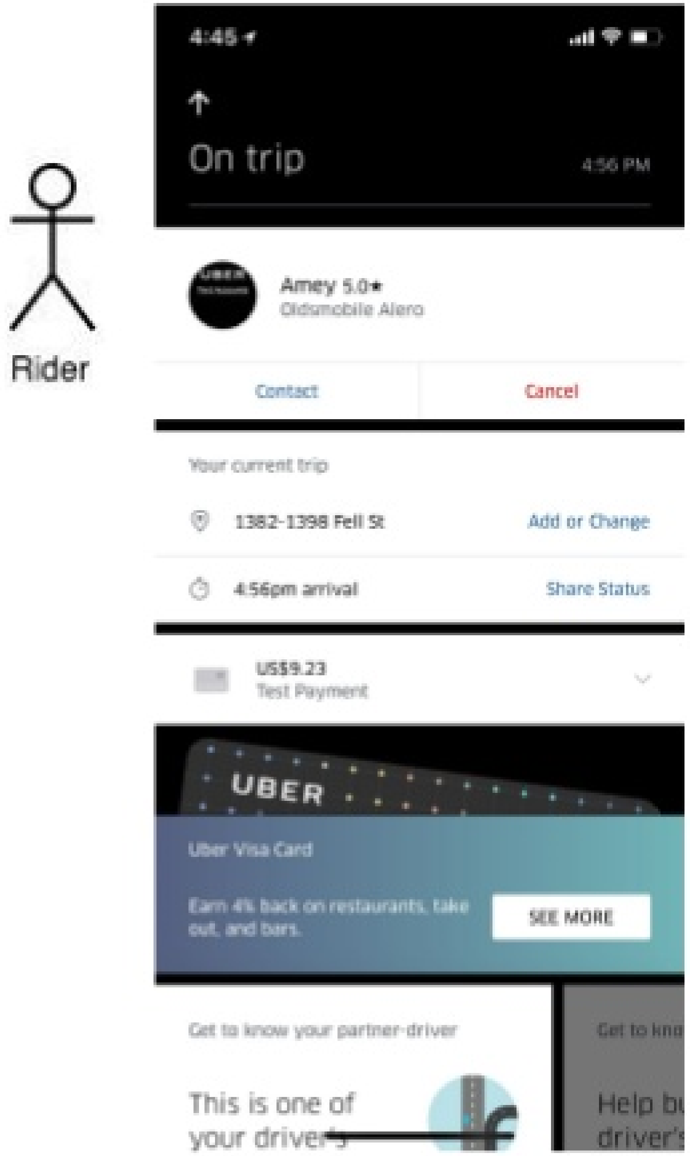
The Sessions State Machine - Cancellation (the sad path ☹️)



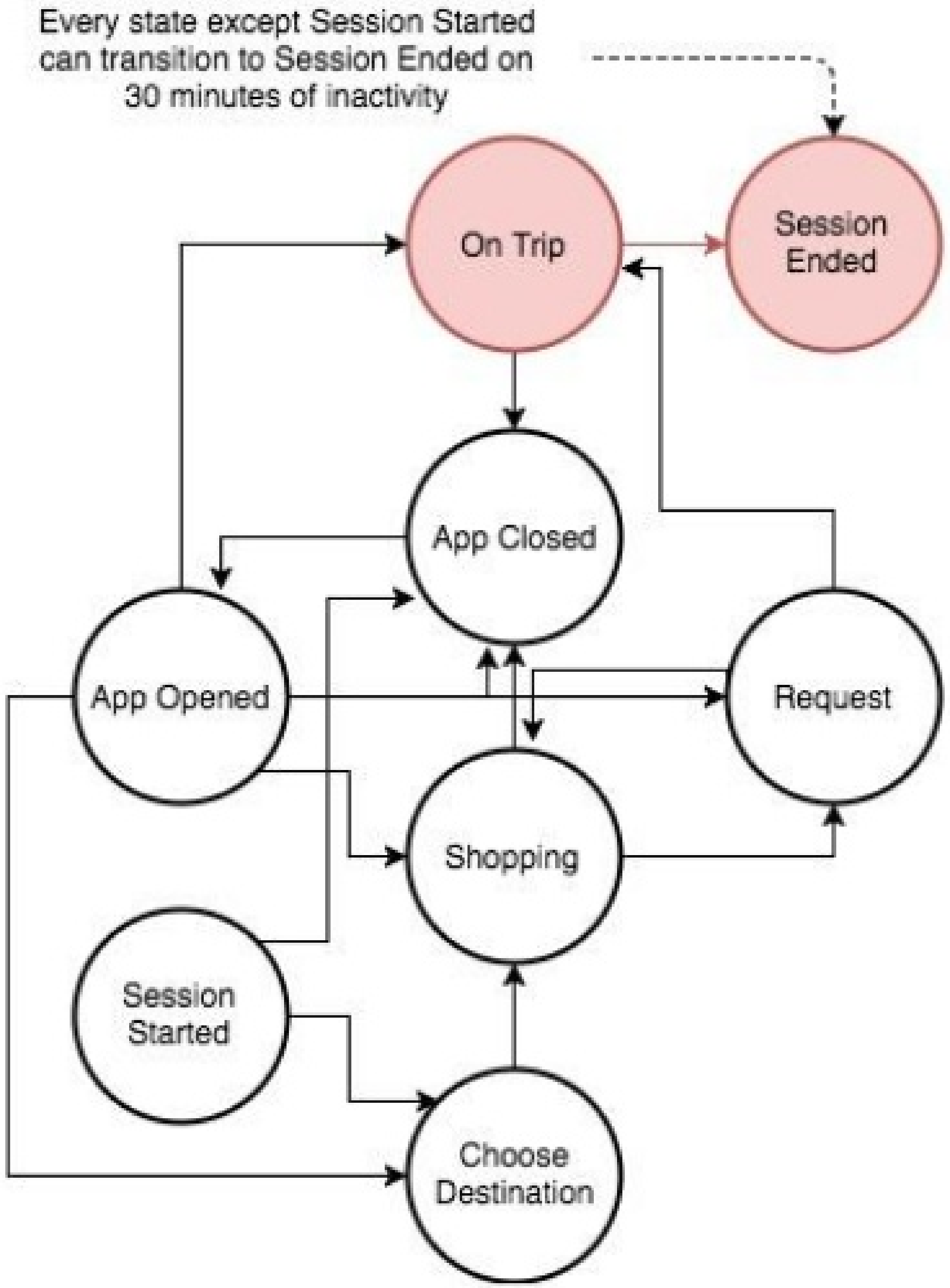


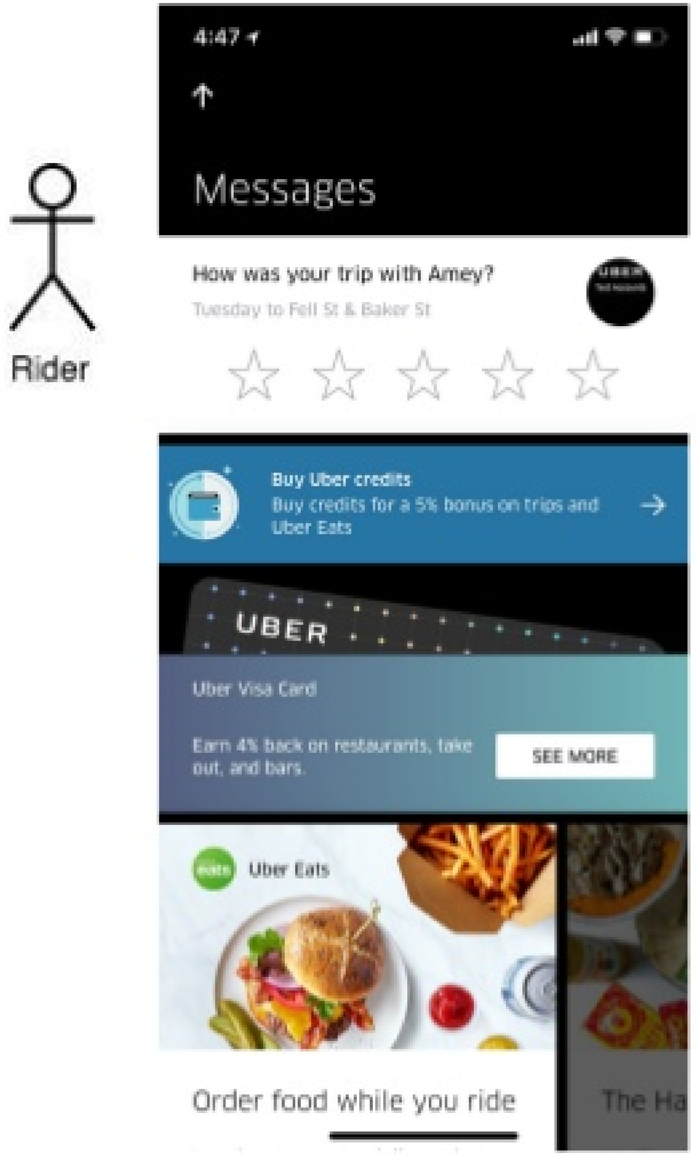
The Sessions State Machine - On Trip (The happy path □)





The Sessions State Machine - Session End (The happy path □)





Implementation

Pickup Experience at Super Bowl 2017

Sessions DSL

```scala
type Condition = SessionInput => Boolean

/**
 * This class defines a single transition. It takes in two functions:
 *
 * @param condition Function of type (SessionInput => Boolean) representing conditions that trigger this transition.
 * @param nextState Function of type (State, SessionInput) => State, which results in the next state.
 */
class Transition(condition: Condition, nextState: (State, SessionInput) => State) {
 def isConditionValid(event: SessionInput) = condition(event)
 def transition(event: SessionInput, fromState: State): State = nextState(fromState, event)
}
```

e.g.

```scala
val requestRideAndRiderLookingTransition = new Transition(isRequestEvent and isRiderLooking, RequestRideState.transitionTo)
```

Sessions DSL

```scala
sealed trait State {
 val ts: Long
 val name: RiderSessionStateName.Value
 val transitionReason: Option[String]
 val jobUuid: Option[String]
 val originLocation: Option[GeoLocation]
 val destinationLocation: Option[GeoLocation]

 // List of Transitions out of this state. They are evaluated in the order of precedence they appear in this list.
 val transitions: List[Transition]

 /**
   *
   * @param event Current session input.
   * @return The next state resulting from the input event.
   */
 def withEvent(event: SessionInput): State = {
   // Find the transition with condition that event holds valid.
   val transition = transitions.find(t => t.isConditionValid(event))
   transition map(_.transition(event, this)) getOrElse this
 }
}
```

Sessions DSL - Putting it all together

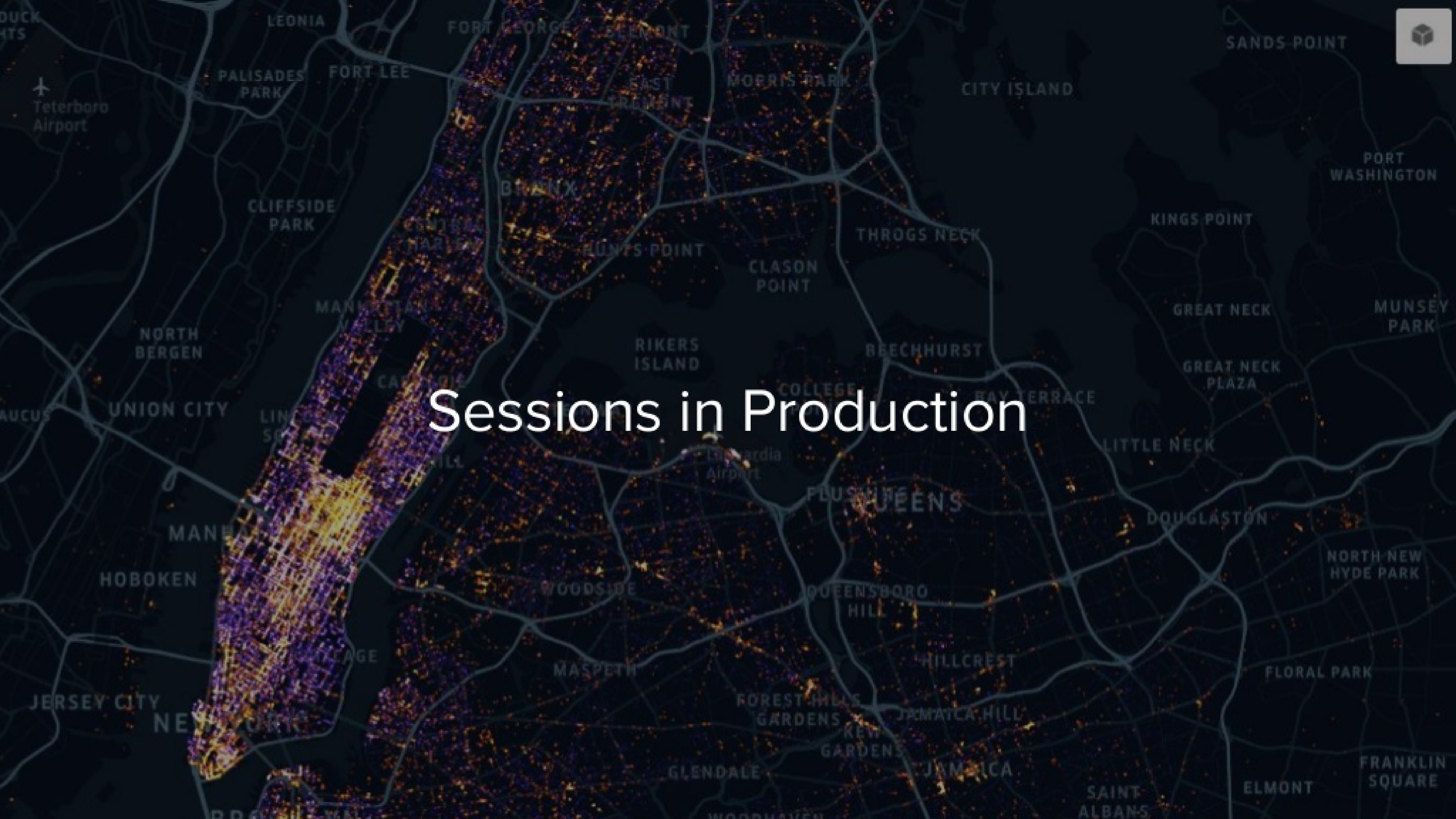
```
case class RequestRideState(ts: Long,
                            override val vehicleViewId: VehicleView,
                            transitionReason: Option[String] = None,
                            jobUuid: Option[String] = None,
                            originLocation: Option[GeoLocation] = None,
                            destinationLocation: Option[GeoLocation] = None) extends State with AssignedVehicleView {
 override val name: SessionState.Value = SessionState.RequestRide
 override val transitions: List[Transition] = List(Transition.onTripTransition,
   Transition.shoppingStateFromRequestTransition,
   Transition.requestRideSelfTransition,
   Transition.shoppingStateOnRequestExpiration)
}
```

Sessions - Moving from Spark to Flink

```
unionedStreams
 .assignTimestampsAndWatermarks(new BoundedOutOfOrdernessTimestampExtractor[SessionInput](Time.seconds(30)) {
   override def extractTimestamp(event: SessionInput): Long = event.timestamp })
 .keyBy(_.riderUuid)
 .window(EventTimeSessionWindows.withDynamicGap(Time.minutes(30)))
 .evictor(DeltaEvictor.of(30000.0D, FlinkSessionsPipeline.deltaFunction, true))
 .process(new ProcessWindowFunction[SessionInput, List[RiderSessionObject], String, TimeWindow]() {
             ...
}
```

Spark's *stateFunc* abstraction just fit nicely into ProcessWindow

Sessions in Production



“Time is relative… and clocks are hard.” - I. Brodsky

Our state machine models interactions between **riders**, **drivers,** and **internal back-end systems** each with their own notion of time!

We essentially need to keep track of **per-key watermarks.**

Checkpointing

Checkpointing to HDFS can be **unreliable**.

What levels of **backpressure** can your **downstream applications tolerate**?

At times, it's just easier to **store per-key state to Kafka** and **restart** a new pipeline, letting backfill take care of any gaps.

Backfills

We rely on upserts into Elasticsearch and backfilling can introduce subtle bugs by being **“more correct.”**

In our implementation each **session is indexed** by an **anonymized rider UUID** and its **start time**, i.e. event time of the first event to kick off a session.

Re-ordering the event consumption in a backfill can easily give you two nearly identical sessions with slightly different start times

Schema Evolution

Uber's ride sharing **products** are **constantly evolving** and our pipeline needs to **keep in sync.**

A new pipeline without a state needs to **warm up the state** before we can trust it to reliably write to our production indices.

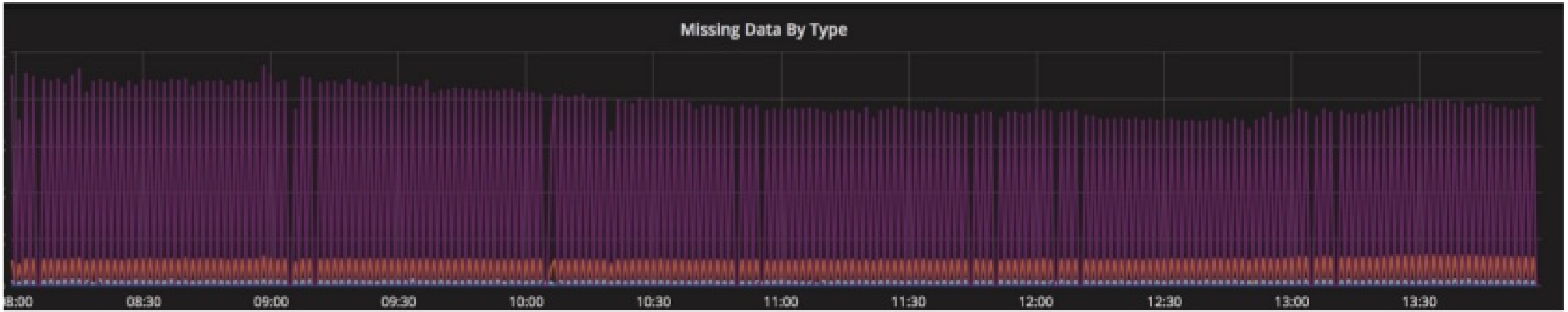
Production hand off between the old and the new pipelines needs to be **carefully choreographed.**

Observability

Keep track of **any** and **every metric** you can possibly think of for **data reliability** as well as processing metrics such as Flink & JVM stats.

We use **M3**, our open source metrics platform for **Prometheus**.

Upstream **data can** and **will change** on you.



Imputing State

Uber's first and foremost responsibility is to ensure a reliable, safe ride for our users from request/dispatch all the way to a completed trip.

Mobile logging is **buffered** and the **best option,** especially when running on low tech phones and in areas with poor network connectivity.

Our sessionization pipelines need to be **resilient to dropped events**

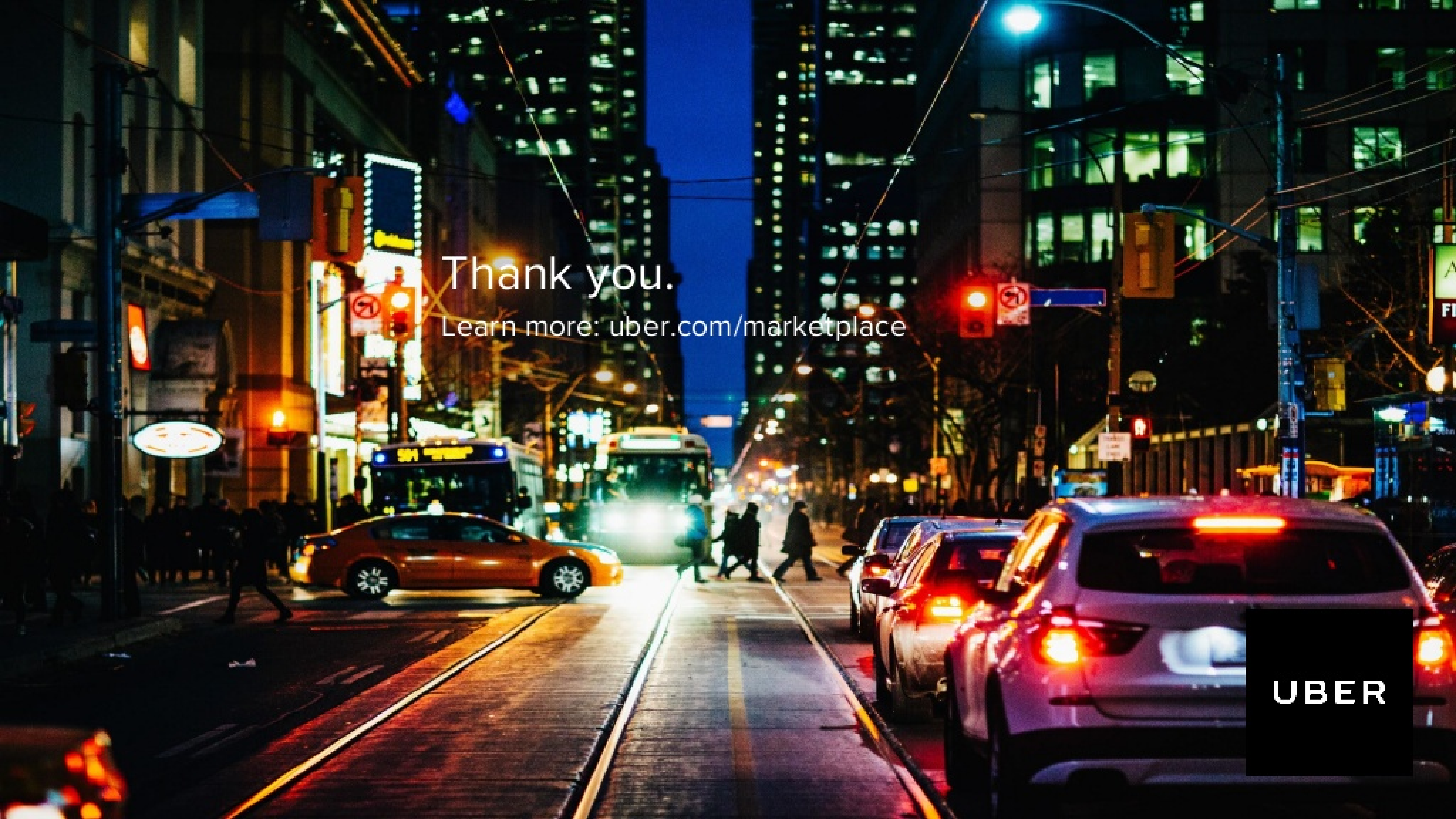
Sessions in Numbers

Scale

We ingest **tens of billions** of events daily.

We generate **tens of millions** of **sessions.**

Currently the production pipeline is running in Spark Streaming and we’re comparing it against a Flink successor.



Thank you.

Learn more: uber.com/marketplace

UBER