Real time Forecasting @ Lyft

Ravi Magham

September 4-5, 2024

Sunnyvale, CA. USA

Agenda

- Context
- Streaming platform
- Real time Forecasting
- Learnings
- What’s Next

Forecasting

Context

- Forecasting is crucial for Lyft to efficiently manage it’s marketplace and ensure optimal service levels.

- Accurate forecasts help align driver availability with rider demand.

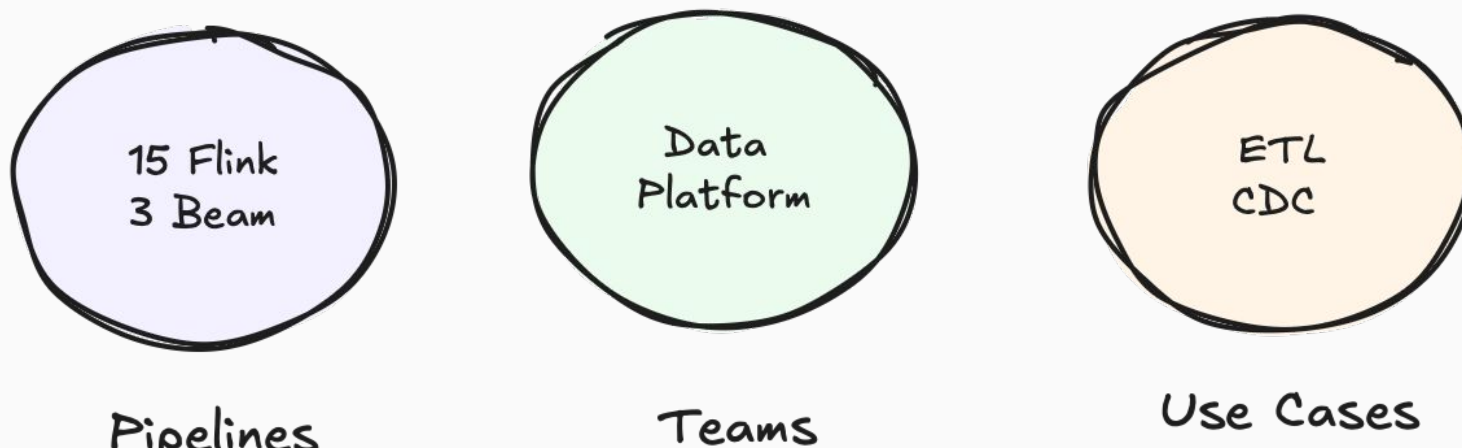
- ML models predict supply & demand in real time **every minute**
  - on ~4 million gh6, +airports +venues
  - for 60 mins horizon in 5 min buckets

- Influences critical Lyft products, eg.
  - Real time Incentives
  - Dynamic Pricing
  - Primetime

Streaming Platform

Circa 2021

- Internal forks of Flink - 1.10, Beam - 2.18



Current

- Internal forks of Flink - 1.17, Beam - 2.50

BEAM SUMMIT

Why Beam

- Flink ++ (like side inputs, x-lang)
- Portability across Runtimes
- API abstraction
- Data scientists productive in Python over Java :)

Deployment Workflow

Client

sp_api svc

Flink Cluster

Control Panel

etcd

api

k8s operator

ing

SVC

Task Manager Pod

Task Manager Container

Executable Stage

Group By Key

Executable Stage

SDK Harness

Job Manager Pod

Job Manager Container

Flink Rest API

FlinkPortable ClientEntryPoint

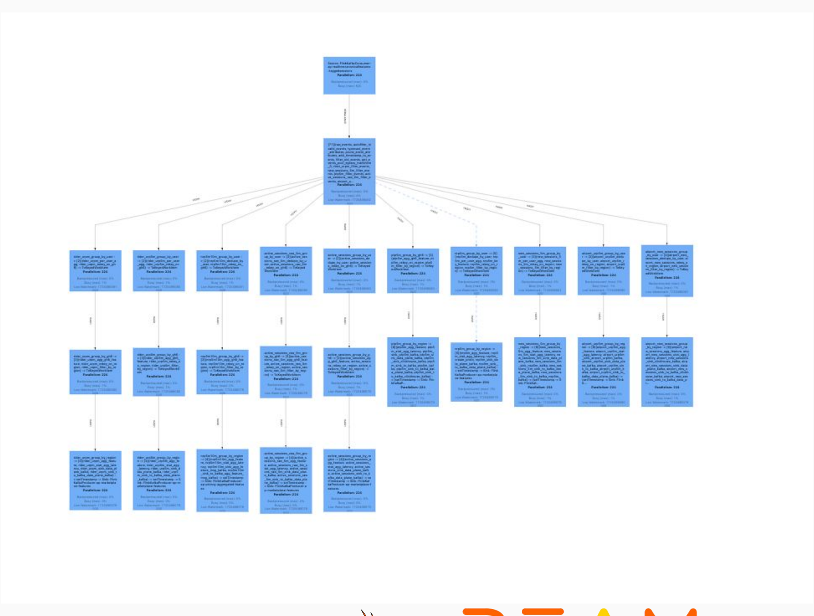
Beam Server Job

Python Entry Point

Architecture

Scale

- Parallelism : 250
- Tasks : 7500 tasks
- Input
  - ~5 million events per minute for supply
  - ~1 million events per minute for demand
- Task Manager(s) : 30
  - Cpu : 24
  - Memory : 192 GB



Architecture - I

Events

Events

Fanner

Kinesis

Filter

Enrich

Windowing

service

Kafka

Key by model

Handler

Handler

fcast

fcast

Model Serving

Kafka

Feature Store

Data ingestion

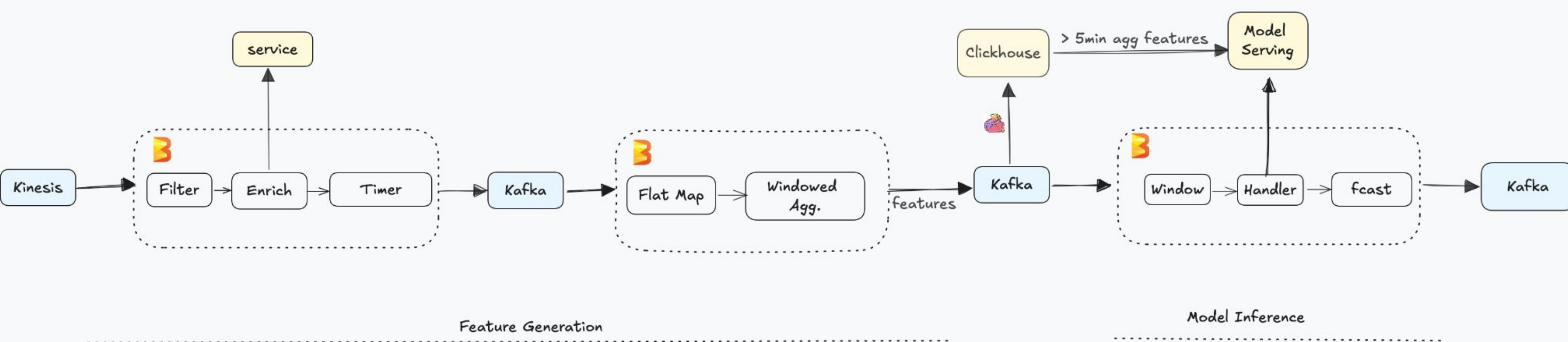
Feature Generation

Model Inference

Challenges

- Feature Generation
  - Out-of-band communications to Services
  - Memory Constraints of Long-Running (> 5 mins) Sliding Window Aggregations
  - High checkpointing times and continuous backpressure.
  - Metastable failures - Long GC, Zookeeper
- Model Inference
  - **N** disparate sinks – throttling
  - Code complexity with **M** models and **R** ⊆ **M** invocation rules.
  - Shadow run models involved launching context-aware shadow pipelines

Evolution



BEAM SUMMIT

Takeaways

- Decouple data prep (filtering, svc enrichment) from feature eng. (windowed aggregations)
- Choose partitioning key wisely to limit skewness
- Filter early, filter often.
- Use efficient coders - protobuf
- Monitor memory and (try) limit large sliding window aggregations.
- BatchElements to amortize costs.
- Cache using shared object

Backfill

- Streaming jobs can fail due to various reasons:
  - Source / sink failures
  - Transient service failures
  - Upstream data changes

- Onboarding new models require bootstrapping features computed from historical time windows

Backfill Options

- Separate Batch job
  - Maintain multiple jobs
  - Recipe for online-offline skewness

- Streaming only
  - Custom PTransform with Kinesis and File Source
  - Flink Global watermarks for source synchronization to avoid state size explosion

```
def create_kinesis_and_s3_input():
    input = S3AndKinesisInput()
    event_config = EventConfig(name='event_intents')
    event_config.with_lookback_in_days(7)
    input.with_event_config(event_config)
    input.with_kinesis_stream_name("stream_name")
    return input
```

Learnings

YAML


```yaml
pipeline_name: intents
sources:
  - type: kinesis
    kinesis_stream_name: demand_events
transforms:
  - type: convert
    transform_name: key_on_id
    events:
      - event_names: [ride_requested, ride_accepted]
        event_transform: demandingestion.functions.key_on_id
aggregation:
  - type: window
    transform_name: initial_windowing
    window_type: sliding
    window_size_sec: 600
    window_period_sec: 60
    trigger:
      type: processing_time
      time_to_wait_sec: 30
  - type: stateful_aggregation
    transform_name: 1min_agg
    cls: demandingestion.aggregators.StatefulAggregation
```

Tools

- Custom FlinkStreamingPortablePipelineTranslator to register Flink Kinesis, Kafka, S3 connectors with configurable parallelism.
- Automatically categorize error logs by User or System
- Validation (quasi canary style deployments)
- Analyze Job scale
- Fault Injection tests

```
> sp --help

Commands:
  validate        Validates a Flink config
  restart         Restarts a Flink Application
  rescale         Redeploys a Flink Application with a new parallelism
  show            Compile a set of configs into YAML
  status          Gets the status of a Flink Application
  update          Updates a Flink cluster with the provided configs
  cancel-deploy   Cancels an in-progress deploy
  teardown        Teardown a particular version of flink cluster
  analyze-scale   Analyzes performance of a Flink application
```

Knobs

- **Job**
  - Network buffers
  - Tune Managed memory taskmanager.memory.managed.fraction > 0.4
  - Checkpoint Local Recovery
  - (Auto) tune CPU and Memory
  - Bundle size, sdk worker parallelism, task slots.

Knobs

- **Cluster**
  - Disallow node eviction by autoscaler with pod annotation
    [cluster-autoscaler.kubernetes.io/safe-to-evict]
  - Use taints & tolerations to pin jobs to single AZ.
  - Address cluster fragmentation through an automated job re-deploys.
  - Disable DNS caching

(Customer) Surprises

- Head-of-Line Blocking
- Autoscaling
- Canary deployments
- Transient Failures ➔ Stop the world ➔ Restart from checkpoint
- Tradeoff in Completeness, Latency and Accuracy

Summary

- Ease of use
- Templatize best practices
- Simple but robust tools
- Integrate into the ecosystem
- Shift left

Plans

- SQL

- Reliability
  - Load shedding
  - Autoscaling
  - Workload Isolation / Cell based architecture
  - Observability into SDK harness grpc chatter

Thank you!

Questions?

@maghamravi

https://www.linkedin.com/in/ravimagham/

https://beam.apache.org/case-studies/lyft/