Architecture of Fennel's Real Time Feature Platform

Nikhil Garg, CEO, Fennel

Organized by

Open Source ML Systems That Need To Be Built

Nikhil Garg

@nikhilgarg28

Quora

1. Model Management
2. **Feature Extraction Framework**

Journey to Starting Fennel

- Quora: Quality/Abuse/Risk ML, followed by ML Platform
- Facebook: Ran group of ~100 ML engineers, saw cutting edge realtime ML in both recsys and risk/integrity
- Ran teams behind PyTorch, saw power of “beautiful” Python APIs
- Even in early 2022, no feature stores existed that lived up to my manifesto from 2017, so decided to build one with ex-Facebook team

Agenda

1. Intro to Fennel Abstractions (Walkthrough)
2. Fennel's Unique Architectural Choices
3. Under the Hood: Streaming Engine

1. Intro to Fennel Abstractions

(Walkthrough)

2. Fennel's Unique Architectural Choices

1. Simplifying Authoring: Pure Python, no DSLs or PySpark

- What: let users write real Python with Pandas, arbitrarily use any of their own Python code or Python ecosystem
- Why: meet users where they are, innovate at lower layers for perf, if needed
- Page from the PyTorch playbook - focus on a) ease of use b) interop with full Python ecosystem
- How: built in Rust, fast Python/Rust Interop by embedding Python interpreter, PEP 684 (per sub-interpreter GIL)

2. Simplifying Realtime: Streaming first via Kappa Architecture

- What: No two separate batch/streaming subsystems. Everything is always streaming all the time, batch is just a special case
- Why: ML is rapidly going realtime, almost everyone has *some* streaming feature use cases
- Removes a ton of complexity and many interaction modes
- **Exact** same code works across batch & streaming
- How: in-house stream processing system written in Rust (vs Spark/Flink)

3. Simplifying Realtime: Read/Write Separation, Features as Functions, Not Data

Write Path
Read Path
Your data sources
Streaming Pipelines
Dataset A
Dataset B
Dataset C
Dataset D
Dataset E
Data connectors
Data Lookups
Feature F3
Feature F1
Feature F4
Feature F2
Feature Dependencies
REST Endpoint
Live Request

4. Simplifying Quality: Native Best-in-Class Quality Tooling

- What: whole suite of primitives for data/feature quality - both preventive & diagnostic
- Why: data/feature quality issues occur far too often, most don't have the tooling to even realize that
- How: add all best practices - versioning, immutability, strong typing, data expectations, unit testing, drift monitoring etc.

Expectation	Successful	Failed
expect_column_values_to_be_between (ad_clicks) Threshold: 95% 100%	10020 rows	1 row
expect_column_values_to_be_between (ad_clicks_7d) Threshold: 95% 99%	13121 rows	101 rows
2 total Expectations		

Fig 1: Data Expectations

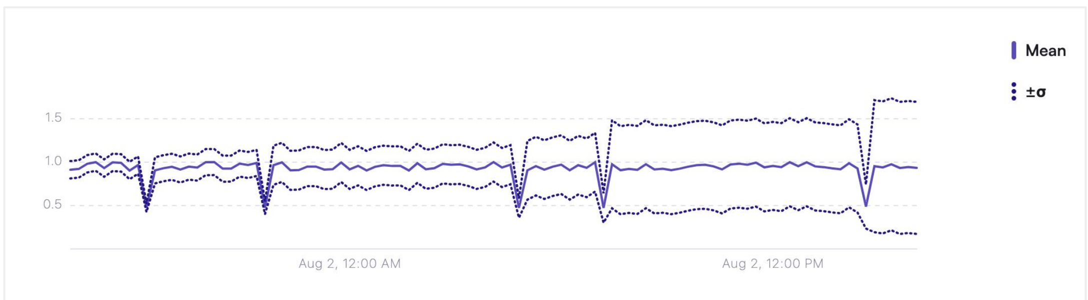

Fig 2: Drift Monitoring

5. Simplifying Ops: Deep Vertical Integration; Anti-Virtual

- What: only commercial feature store to not require customers to bring own K/V store or compute engine or metadata store etc.
- Why: lots of ops & cost issues in running each component. Much better experience AND ops AND cost via vertical integration. Shared dependencies also leads to operational confusion (e.g. who should autoscale a shared cluster?)
- How: Fennel brings up and manages everything that is needed. Zero dependency installation.
- Makes our own surface area much wider/harder but better for customers.

6. Simplifying Infosec: Thick Data Plane, Thin Control Plane

- What: like other modern data tools, Fennel is deployed as data plane inside customer cloud with a single control plane. Unlike others, data plane is thick enough to be nearly self-sufficient.

- Why: thick dataplane can handle full lifecycle of features ⇒ user data & feature code NEVER leave dataplane in the customer cloud

- How: Dataplane has all the machinery / services to be self-sufficient. Even console UI runs inside data plane so feature code doesn't have to leave customer cloud. Control plan for only new code deployment and telemetry (and could go down without affecting any live traffic).

3. Under The Hood: Streaming Engine

No Sync Communication, Full Shared Nothing Architecture

- Flink operators communicate with each other via RPC calls, even when deployed across machines ⇒ one node going down interrupts working of other nodes
- Fennel doesn't do any sync communication across nodes. Either comms are within a node (so in-memory) or async mediated via Kafka.
- Kafka ⇒ natural handling of backpressure, backfilling etc. Nodes can go down without affecting other nodes
- Also gives exactly once processing "for free"

Management of State of Streaming Jobs

- Every streaming job is passed a handle to KV store implementing a generic interface (default RocksDB)
- Writes during an iteration are “buffered” in memory and applied together later in a batch atomically (in sync with exactly once transaction of input/output Kafka topics)
- Periodic backups (using EFS) of state store every few minutes
- Transaction markers for state store also stored in Kafka for quick recovery on machine failures

Continuous Distributed Checkpointing

- Flink's checkpointing model is "stop the world" which doesn't work when stream topology changes often (e.g. when new features are written)
- Each job checkpoints itself independent of other jobs.
- Local SSD checkpoint - every second or so. Object store checkpoint: every few minutes
- Benefits: Any node of the system can go down any moment without creating any issue + fast recoveries

OOMs are Structurally Impossible!

- Spark (in particular) and Flink (to some degree) have a tendency to run out of memory for feature engineering jobs
- With Fennel's stream engine, no in-memory state of jobs – state lives on disks (storage systems often keep it warm in block/page cache, so not very costly to read/write either)
- No in-memory shuffle operations either. All shuffle is mediated via Kafka
- No GC (thanks to Rust) & very tight control on RAM ⇒ lower costs because memory is often the costliest component

Streaming Engine Is Time Aware!

- Unlike Spark/Flink, engine is aware of data schemas and each schema must have a unique event timestamp ⇒ engine is aware of event timestamps
- Natively handles watermarking, out of order handling, time window aggregations, etc.
- Enables point-in-time correct streaming joins, which is *very very* hard to get right with good efficiency

No Central Scheduler; Horizontally Scalable

- No central scheduler - each worker node has its own “supervisor” that is responsible for some shard space
- Can horizontally scale by having higher number of shards and/or more supervisors
- Another design choice to avoid sync communication & single point of failure

Thank you! Questions?

fennel.ai/docs

Nikhil Garg
nikhil@fennel.ai