Powering Real-time Data at Intuit: A Look at Golden Signals powered by Beam

By Dunja Panic, Nick Hwang, Omkar Deshpande, & Nagaraja Tantry
Agenda

Intuit’s Stream Processing Platform

Developer Experience

Platform Architecture

Featured Use Case

Q&A
Who we serve
Consumers
Small businesses
Self-employed
Powering Prosperity with AI and Data-driven platforms

Intuit customers want to get their issues resolved in the most efficient way possible to feel confident in their outcomes. We want to Intelligently route users to the right expert to quickly resolve their issue.
## Origins of Stream Processing Platform

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Adhoc workloads with low data freshness</td>
<td>• High data availability and data freshness</td>
</tr>
<tr>
<td>• High infrastructure costs across teams</td>
<td>• Cost savings due to shared infrastructure</td>
</tr>
<tr>
<td>• Custom integrations handled on an adhoc basis, team by team</td>
<td>• Standardized integrations with Intuit developer ecosystem</td>
</tr>
<tr>
<td>• Team focused on operating streaming infrastructure</td>
<td>• Teams focus on producing/consuming clickstream or application events</td>
</tr>
</tbody>
</table>

- **3x** Improvement in Speed to Market
- **5x** Reduction in cost
- **240x** Improvement in data availability
Data Engineer

“I want to focus on rapidly developing streaming applications so that I can provide real-time personalized user experiences in my product”
Key Features

- **Push-button** pipeline management
- Completely **managed** infrastructure
- **Out-of-the-box** starter code and dashboards
- Programming language and processing execution engine **flexibility**
- Rich **discoverability** and exploration of Intuit data ecosystem
Developer Experience on SPP
Processors and Pipelines

- Processor = Business Logic & Code
- Pipeline = Deployment & Infrastructure

- Serial processors (e.g., reusable intermediate topic)
- Parallel processors (e.g., fleet deployment)
DevPortal: Intuit’s home for “self-serve paved roads”
With a couple clicks...
Everything you need to start coding

STREAM PROCESSOR

Edit Base Asset

Stream Processor Info  Manage Processor

Select New Jenkins Master

Source Language Type: Java
Visibility: Project
Framework: javabeam

Created: Thu Jul 30 2020 01:33:48 GMT-0700 (Pacific Daylight Time)
Last Updated: Thu Jul 30 2020 01:33:48 GMT-0700 (Pacific Daylight Time)

- Sample Stream Processor Github Repo:
  https://github.intuit.com/data-processingpt/sessionization

- Build Url:

- Artifactory Url (release):

- Artifactory Url (snapshots):
Locally iterating until cloud-ready
Processor = Code
Pipeline = Cloud Deployment
With a couple clicks...
All the infrastructure you need

STREAM PIPELINE

Env: QAL
Region: US-WEST-2
Runner: Flink
IKS Version: IKS2

Pipeline Status: STOPPED
Namespace: data-processingpt-ecsstatefuliks2-usw2-qal
Cost Dashboard: QlikSense
Splunk Logs  Wavefront Dashboard  MDR Properties  MDR Lineage

Flink  Amazon S3
kubernetes  AWS

Author  Compose  Deploy
Compose your processors...

<table>
<thead>
<tr>
<th>Action</th>
<th>Name</th>
<th>Description</th>
<th>Resources</th>
<th>Last Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>CPUs: 4</td>
<td>12:30:37 GMT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Mem: 4Gi</td>
<td>0700 (Pacific)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Offset: 1</td>
<td>Daylight Time</td>
</tr>
<tr>
<td>Delete</td>
<td></td>
<td></td>
<td>History</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vep-engagement-integration</td>
<td>Processor: Intuit.expertnetwork.engagement.service, rev: 37</td>
<td>Replicas: 3</td>
<td>Thu Jul 07 2022</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CPUs: 4</td>
<td>12:30:37 GMT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Mem: 4Gi</td>
<td>0700 (Pacific)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Offset: 1</td>
<td>Daylight Time</td>
</tr>
<tr>
<td>Delete</td>
<td></td>
<td></td>
<td>History</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vep-int-match-making</td>
<td>Processor: Intuit.expertnetwork.engagement.service, rev: 44</td>
<td>Replicas: 3</td>
<td>Thu Jul 07 2022</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CPUs: 4</td>
<td>12:30:37 GMT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Mem: 4Gi</td>
<td>0700 (Pacific)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Offset: 1</td>
<td>Daylight Time</td>
</tr>
<tr>
<td>Delete</td>
<td></td>
<td></td>
<td>History</td>
<td></td>
</tr>
</tbody>
</table>
...fine-tune to your heart’s content...
...and get ready to click the big green button!
Watch your deploy complete...
...see your pipeline’s status...
…look for exceptions in a haystack…
…monitor your pipeline’s performance…
...and set up alerts
Multiple entry points to the platform

Our native web experience

Our API
(e.g., Gitops, ad hoc scripts)

Third-party apps using our API
(e.g., feature processing, stream materialization)
Stream Processing Platform
Tech Stack
## Tech Stack Overview

**Customer Experience**

- Application Layer
- Processor CI/CD Layer
- UX Layer
- Control Layer
- Pipeline CI/CD Layer
- Runtime Layer
- Infrastructure Layer

**Behind-the-scenes**

- beam
- GitHub
- Jenkins
- JFrog Artifactory
- React
- Splunk
- Wavefront
- Qlik
- spring
- Amazon Aurora
- Jenkins
- JFrog Artifactory
- Argo
- podman
- Flink
- RocksDB
- Amazon S3
- Kubernetes
- AWS
Application Layer

Components
- SDK libraries

Core functions
- Auto Kafka configuration
- Data access policy handling
- Metrics collection

Guiding principle: Developer flexibility
Runtime Layer

Customer Experience

Behind-the-scenes

Application Layer
Processor CI/CD Layer
UX Layer
Control Layer
Pipeline CI/CD Layer
Runtime Layer
Infrastructure Layer
Runtime Layer

Guiding principle: Scalability

Components
- Flink application cluster
- S3 for fault tolerance

Core functions
- Stateful processing support
- Fault tolerance and at-least-once processing
- Low deploy/restart latency
- Health metrics
- Highly tunable and configurable via UX Layer
- Auto-scaling
Infrastructure Layer

- Application Layer
- Processor CI/CD Layer
- UX Layer
- Control Layer
- Pipeline CI/CD Layer
- Runtime Layer
- Infrastructure Layer

Customer Experience

Behind-the-scenes
Infrastructure Layer

Components
- Kubernetes clusters on AWS EKS

Core functions
- Namespace isolation
- Low deploy/restart latency
- Rich operational metrics
- Fault tolerance
- Billing tags
- Multi-cluster topology

Guiding principle: Multi-tenancy
Deployment workflow

- Container - Build beam + flink application image
- Provision compute - instance group
- Provision state storage - S3 bucket
- Configure IAM role permissions
- Generate configurations - e.g. Kafka broker addresses
- Create the Kubernetes objects
- Capture lineage
Anatomy of a flink application on Kubernetes
What’s underneath?
AWS components

- Amazon S3
- IAM role
- Kubernetes addons
- Auto Scaling group
  - Amazon EC2
  - EBS
- Load balancer
- VPC
- Amazon EKS
Learnings

- **Runner migration**
  - Samza -> Flink

- **Multi tenancy model**
  - Disruptions caused by scheduler
  - Disk isolation
Summary & Learnings

Guiding Principles

- beam  Developer flexibility
- Flink  Scalability
- kubernetes  Multi-tenancy

Lessons Learned

- **Having runner flexibility can be really nice**
  - We changed our runtime from Samza to Flink, and customers didn’t have to write any new code

- **Compute isolation issues can surprise you at scale**
  - Pod disruptions caused by k8s scheduler made full multi-tenancy tricky to stabilize
  - Lack of disk isolation can become a performance bottleneck
## Summary & Learnings

<table>
<thead>
<tr>
<th>Tech Stack Layer</th>
<th>Core Technology</th>
<th>Guiding Principle</th>
<th>Lessons Learned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>beam</td>
<td>Developer flexibility</td>
<td>Runner flexibility allowed us to change runtime from Samza to Flink</td>
</tr>
<tr>
<td>Runtime</td>
<td>Flink</td>
<td>Scalability</td>
<td>Flink experiences processing disruption if k8s scheduler is overly aggressive</td>
</tr>
<tr>
<td>Infrastructure</td>
<td>kubernetes</td>
<td>Multi-tenancy</td>
<td>Lack of disk isolation on k8s resources can become a performance bottleneck</td>
</tr>
</tbody>
</table>
Featured Use Case
Golden Signal for Services
### Service Golden Signals

<table>
<thead>
<tr>
<th>System defined</th>
<th>Opinionated Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Availability</strong></td>
<td><strong>Health</strong></td>
</tr>
<tr>
<td>Success Rate of Service calls.</td>
<td>Health of an application or service in real-time. May be redefined by application teams. Typically based on aggregate availability.</td>
</tr>
<tr>
<td><strong>Requests</strong></td>
<td><strong>Saturation / Utilization:</strong></td>
</tr>
<tr>
<td>Measure of demand / load being placed on the system.</td>
<td>How &quot;full&quot; a service is. Percent of “max capacity” being used. Varies by service constraints. e.g. nodes, memory, CPU, networking, auto-scaling limits, etc.</td>
</tr>
<tr>
<td><strong>Errors</strong></td>
<td></td>
</tr>
<tr>
<td>Rate of requests that are failing. (e.g. HTTP 500s)</td>
<td></td>
</tr>
<tr>
<td><strong>Latency</strong></td>
<td></td>
</tr>
<tr>
<td>Time it takes to service a request. Typically measured across percentiles.</td>
<td></td>
</tr>
</tbody>
</table>
High level Design
Customization using Side Input

- An additional input that your DoFn can access each time it processes an element in the input PCollection
- Health metrics and few tags are overridable

Health metrics example

```
times (15 SC) | 72 bytes
1 requestCount < 60 ? 0 : 2 availability > 99.0 ? 0 : 3
```

Swimlane example

- Users use the GitOps model to customize their service
- Override configuration is stored in S3 (Gitops → Jenkins → Upload to S3)
- Pipeline fetches from S3 every 5 min using Beam Side Input
Router Component helps to
- Achieve SLA of 3 min
- Zero Downtime deployment

Controller - Sends a message to worker to flip topics when health check fails

Workers: Reads from source and publishes to destination topic
Q&A