Joseph Spens - Case Studies

I've worked on a number of systems over the years, and each system classification comes with its own joys and challenges. The purpose of this document is to highlight a set of challenges and how the chosen system design addressed those challenges given the available resources.

Application Data Model - *Level*

At Level I designed an application data model for our customer experience teams that decoupled them from our backend data teams, **facilitated faster iteration of feature development** for the application teams, and **reduced risk of action at a distance with schema changes** for our backend data teams. This data model was identified through novel approaches to defining application features, and encoded into backend **protobuf APIs**. I then **embedded in backend teams** to push the technical debt down the stack into the data layer where a new **data model directly supported our application use cases**.

Rights Management System - *Spotify*

At Spotify I designed a rights management system meant to consolidate all the recording and publishing rights known to Spotify about its catalog of **80 million tracks** and **derive licensing and market insights** for our litigation and licensing teams. We derived the initial comprehensive music domain model for our organization, prototyped multiple entity resolution systems with **Dataflow pipelines** in **Scala**, built resilient **React web applications** in **TypeScript**, layered **gRPC services** in **Java**, and robust **PostgreSQL** and **ElasticSearch datastores**.

Background

The litigation team at Spotify is responsible for responding to litigation claims made against the streaming catalog by alleged rights holders of the catalog content. In order to determine the validity of those claims, the litigation team (non technical lawyers) data dive across many disjoint datasets of unstructured raw data ingested from third party providers. This process takes, on average, about **7 months** to assemble sufficient data to

respond to claims, and this data is almost always untrustworthy and contradictory (multiple rights holders each claiming 800% ownership).

Spotify created a company *bet* (a highlighted company objective defined in its Rhythm planning system) to design a tool for the litigation team to surface these licensing insights automatically.

Challenges

1. Create a **centralized and canonical** understanding of of rights information scattered across the company
2. Identify rights for content at the **scale** of Spotify's catalog, which at the time contained around **80 million musical tracks** (recording) described by **hundreds of millions** of fragmented musical works (publishing)
3. Align the signals we collected from **disparate sources** around a **shared domain model** to derive emergent insights
4. Process this information across a **historical context** to understand what Spotify knew at a specific date in the past
5. Over time we **acquired new customers** on the licensing team, whose use cases involved using licensing and market insights to negotiate contracts with rights holders

Solutions

To address these challenges we assembled a number of **Dataflow pipelines** to pull data together into centralized **Avro datasets**, which we used to populate an **Elasticsearch datastore** and a **PostgreSQL database**. That effort required so much resource investment we ended up creating a new squad from that workstream solely to manage publishing ingestion. We added some **Java gRPC services** to vend the ingested data to a **React web application** used by our licensing and litigation teams.

A key feature of this system was deriving a *licensing* insight about our content. Given a musical track, a date, and a territory, to what degree was that track licensed in that territory on that date? Our approach was to introduce an **event based** construct called a *licensing event* which represented a change in the licensing of a track on a specific date, which we derived in our pipeline jobs. With that new entity added to our domain model, we were able to quickly and efficiently build **timeline visualizations** and **search capabilities** to visualize the change in licensing over time.

The system we built shortened the time litigators spent data diving **from 7 months to 24 hours**.

Our investment into a unified **domain model** across recording and publishing domains laid the foundation to what would eventually evolve into a more broadly used music ontology and Spotify's first **knowledge graph**.

Content Annotation Platform - *Spotify*

I also designed a content annotation platform to **platformize access to ground truth training and evaluation data** for machine learning engineers and data researchers, to which we successfully provided **tens of thousands of human annotations** each month. We derived the initial comprehensive human-in-the-loop domain model for our organization, facilitated **ETL processes** through **Flyte pipelines** in **Python**, and enabled rapid development iteration of highly precise **machine learning models** through **active learning**, **transfer learning**, and measuring **training dataset quality**.

Background

Investment in machine learning has been a core company strategy for years, and one of the largest impediments to quick and effective model development (not to mention monitoring once in production) is the availability of ground truth datasets for training and evaluation.

The Content Intelligence product area consisted entirely of squads building binary classification ML models for entity resolution in order to produce canonical datasets for our novel knowledge graph. The availability of ground truth datasets for the models in our domain were particularly sparse, so we were chosen to adopt the mission to **platformize access to ground truth training and evaluation data**, initially in service of squads within our product area as a prototype. The long term goal would be to provide the same ease of access to annotations for other use cases across the company including, but not limited to, audio intelligence, content moderation, personalization, and podcast advertising.

Challenges

1. **Simplify and streamline** the access to human annotations for customer teams
2. Create a **shared understanding** of common annotation needs across machine learning and data research teams

3. **Measure the quality** of annotations and deliver **high quality annotations** to models needing ground truth datasets
4. Provide tools to annotator managers to efficiently **load balance annotators** across annotation projects

Strategies

We were a small team and identified at the start that we're not in the business of building and maintaining annotation tools. The domain is complex and there are many solutions in the industry.

One day our journey might have led to an in-house annotation tool, but for the foreseeable future our mission could be achieved through acquisition of managed annotation tools. For that reason, we adopted a key strategy of **buy over build**.

We were in early discovery of identifying customer personas and use cases, therefore without specific annotation tool requirements, so we also adopted the strategy to **remain tool agnostic**.

Our third strategy was **platformization**, which was identified explicitly in the mission and aligned with broader company strategy.

Solutions

These strategic principles guided us towards prioritizing the facilitation of unlabeled data in and unlabeled data out of annotation tools, and we adopted an initial set of OKRs to define facilitation success. We implemented a set of **Flyte pipelines** in **Python** to move data in and out of our annotation tools, with some light pre and post-processing for our customers. Customers could manage these **ETL workflows** by interacting with an **API gateway** through one of several **client libraries**, a pipelines library for model developer customers and a **command line interface** for model researcher customers.

Our next set of objectives measured ways for us to improve the quality of annotations, which we achieved through educating our customers in unlabeled data sampling, annotation task design, and annotator agreement heuristics. **Customer education and guidance** was a low risk, low cost **proof of concept** that informed future platform capabilities.

To serve our third customer persona, annotator managers, we added pipelines to feed our annotation data into **Looker Studio** dashboards and created **data visualizations**. Those

visualization platform features derived insights into annotator workflows and annotation quality that would inform annotator managers when assigning annotators to projects.

Snapshot Ingestion System - *Ticket Evolution*

At Ticket Evolution I designed a **snapshot ingestion system** using an **event-driven microservice architecture**. This system scaled up our capabilities to **ingest 1 billion daily records** and alleviate the top risk to the business of skipped snapshots and stale data on the platform. The new **Ruby microservices** were designed to use a **Redis cache** instead of the main **Postgres database**, which enabled them to **horizontally scale indefinitely** using an **AWS elastic load balancer**. Creates, updates, and deletes were sent back to the **Rails monolith** through **SQS queues** to be written directly to the database.

Background

Ticket Evolution is a B2B marketplace for brokers to buy and sell live event tickets in groups. The platform ingested over **1 billion** ticket groups in batched snapshots of inventory at regular intervals, the interval depended on the broker, performed over **2.5 million** price updates daily, and its API handled **150 requests per second**. The batch ingestion involved a broker sending us a file containing a **snapshot** of the broker's current inventory. Our platform reads that file and makes the corresponding changes to the current inventory represented in the database.

The platform was designed as a **Ruby on Rails** monolith, a single deployable artifact that did all the processing and read/wrote to the **Postgres** database. This means the same service that supported the end user application interfaces also managed the **heavy batch ingestion processing**. This processing also included reading from the database for each file in order to **diff the snapshot** against the latest information.

Challenges

1. A limited database **thread pool** limited the degree to which we could **scale up our service nodes**.
2. Our ingestion process **throttled our service infrastructure**, tied up **database connections**, and negatively impacted **unrelated web application performance**.
3. The inventory live on the platform **became stale and out of date**, which was the top risk for the company.

4. **Active Record side effects** used for monitoring ingestion and creating alerts for the customer support team are extremely **complex**, resulting in the **inability to predict** what alerts would be created for a given ingestion

Solutions

My first priority was to disconnect the ingestion process from the database, which was the primary bottleneck to scaling the service and handling the increased throughput. I designed an **event-driven microservice architecture** using **AWS EC2, ELB, and SQS**. The microservices were greenfield services written in **Ruby without Rails**, and would communicate with the legacy Rails monolith through the event queues.

This solution was designed to **roll out in parallel** to the existing ingestion system, where ingestion files are processed the old way *and* sent to the new microservices. Instead of the monolith writing the new system updates to the production database, it would log them, enabling us to verify consistency between systems and measure performance. Once we were satisfied with the performance, I began cutting over customers in groups to use the new system, then deprecated the old ingestion process.

Microservice

The entrypoint for receiving inventory snapshots remained unchanged, it instead sent the snapshot files to an inbound SQS queue, where a load balancer (**ELB**) would scale up the new microservice nodes indefinitely to handle the load.

The microservice is responsible for processing the file, which involves sharding the file into multiple sections and processing each in parallel. It uses a number of hashing and diffing strategies, and interacts with a **Redis cache** which contains the previous snapshot. Once the differences in a file are identified, the resulting CUDs (creates, updates, and deletes) are streamed into an outbound SQS queue and processed by the Rails monolith.

Monolith

The Rails monolith then reads in the CUDs and writes them directly to the database. The new database writes for ingestion are very simple and performant, **eliminating additional load** on the database beyond normal operational use.

The big challenge came with processing the inventory changes into customer support *alerts*. These alerts are the result of many independent rules applied against each change to the inventory, and are meant to highlight potential fraud or any other notable change. The rules are complex and the resulting alerts are unpredictable, neither the engineering,

product, or customer support teams could confidently predict which alert(s) would be created from an inventory snapshot.

To fix the alerting issue, which was holding up full rollout of the new system. **I worked directly with the lead customer support rep** and walked through each rule to **refactor the side effects into explicit and colocated rules**, resulting in a complex yet **predictable process**.

Administrative Operations Platform - *Ticket Evolution*