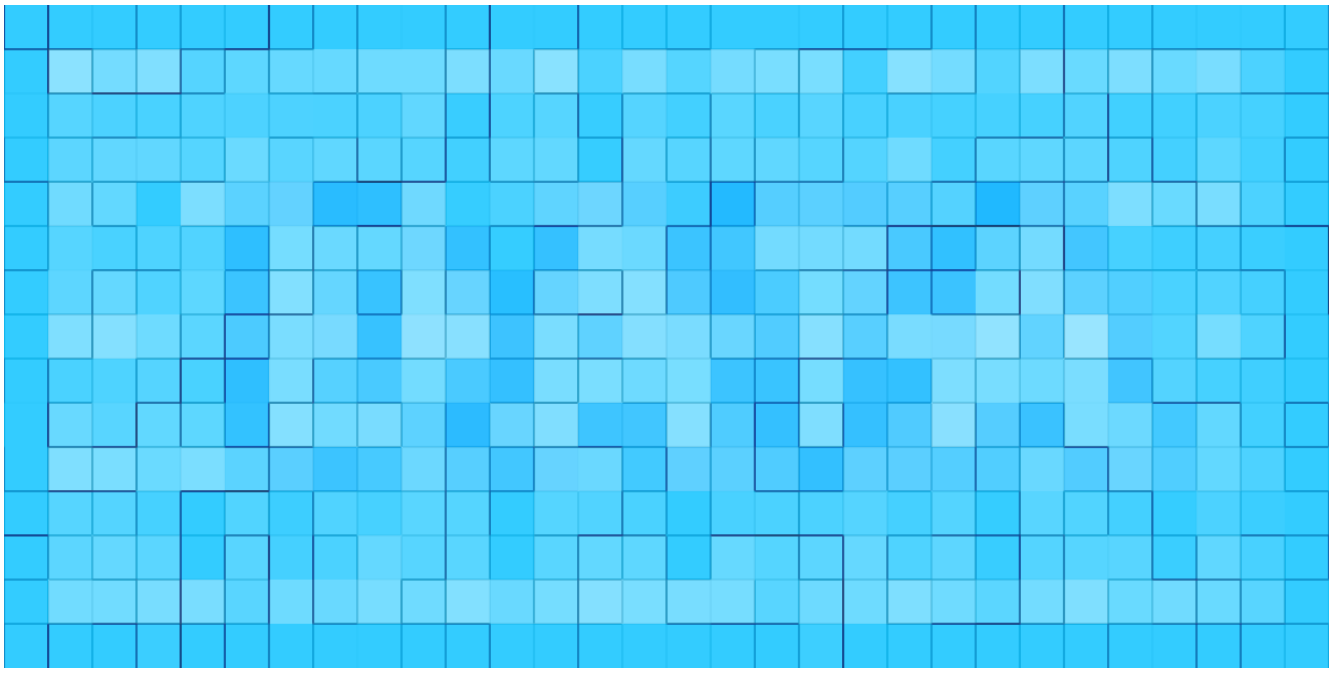
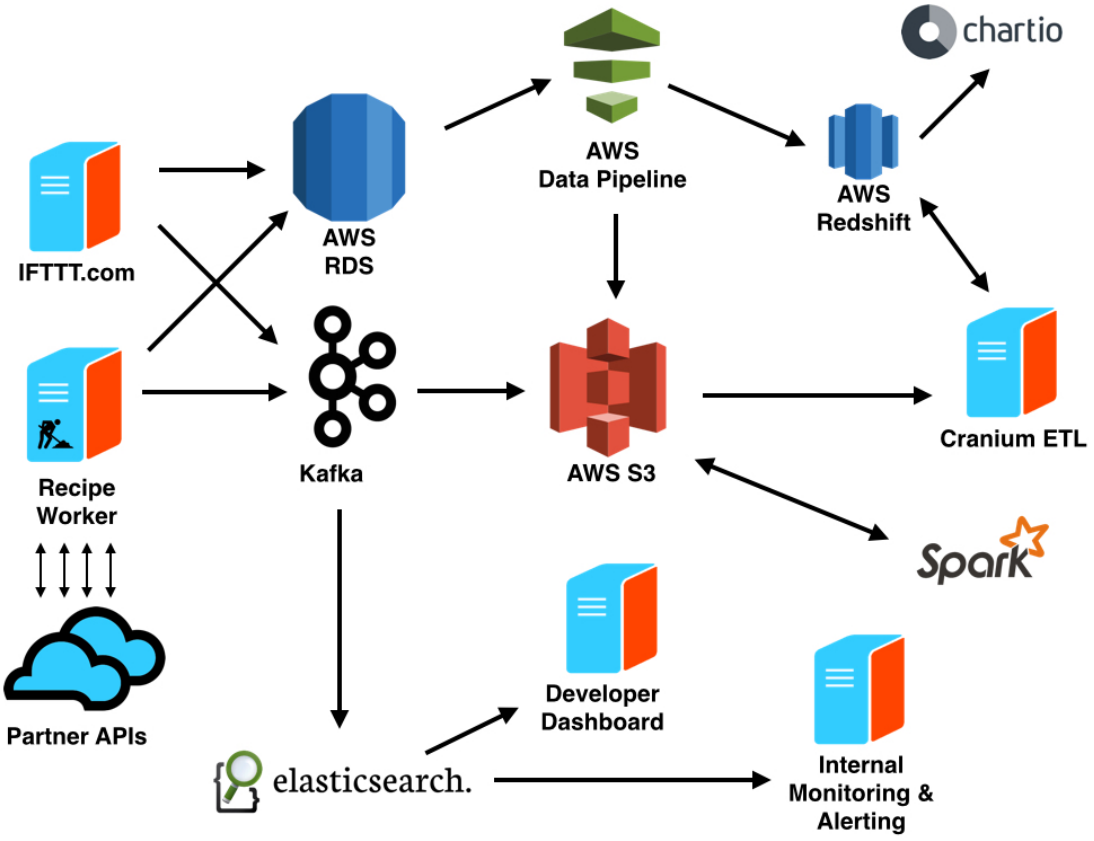
Data Infrastructure at IFTTT

Anuj Goyal - Oct 14, 2015



Data is a big deal at IFTTT. Our business development and marketing teams rely on data to make critical business decisions. The product team relies on data to run tests, to learn about how our products are used, and to make product decisions. The data team itself relies on data to build products like our Recipe recommendation system and tools for spam detection. Furthermore, our partners rely on data to get insights and alerts about the performance of their Channels in real-time.

Since data is so critical to IFTTT, and given that our services generate billions of events per day, our data infrastructure must be highly scalable, available, and flexible enough to keep up with rapid product iteration. In this post, we'll walk you through a high level overview of our data infrastructure and architecture. We'll also share some of the insights we've gained building and operating data at IFTTT.



Data Sources

There are three sources of data at IFTTT that are crucial for understanding the behavior of our users and performance of our Channels.

First, there's a MySQL cluster on AWS RDS that maintains the current state of our primary application entities like users, Channels, and Recipes, along with their relations. IFTTT.com and our mobile apps run on a Rails application, backed by this instance. This data gets exported to S3 and ingested into Redshift daily using AWS Data Pipeline.

Next, as users interact with IFTTT products, we feed event data from our Rails application into our Kafka cluster.

Lastly, in order to help monitor the behavior of the hundreds of partner APIs that IFTTT connects to, we collect information about the API requests that our workers make when running Recipes. This includes metrics such as response time and HTTP status codes, and it all gets funneled into our Kafka cluster.

Kafka at IFTTT

We use Kafka as our data transport layer to achieve loose coupling between data producers and consumers. With this type of architecture, Kafka acts as an abstraction between the producers and consumers in the system. Instead of pushing data directly to consumers, producers push data to Kafka. The consumers then read data from Kafka. This makes adding new data consumers trivial.

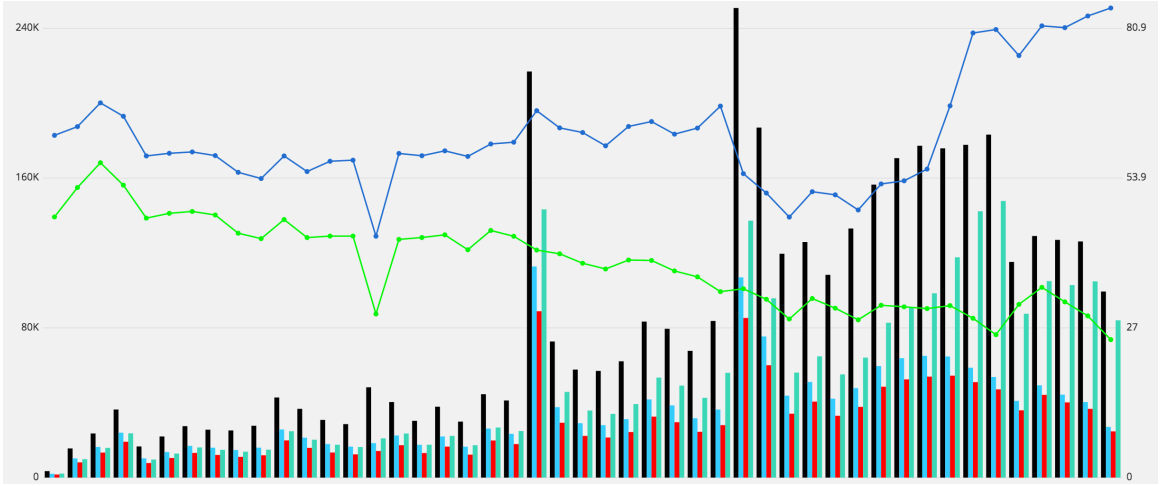
Because Kafka acts as a log-based event stream, consumers keep track of their own position in the event stream. This enables consumers to operate in two modes: real-time and batch. It also allows consumers to reprocess data they have previously consumed, which is helpful if data needs to be reprocessed in the case of an error.

Once the data is in Kafka, we can use it for all types of purposes. Batch consumers send a copy of this data to S3 in hourly batches using Secor. Real-time consumers push data to an Elasticsearch cluster using a library we hope to open source soon.

Business Intelligence

The data on S3 gets ingested to AWS Redshift after transformation and normalization using Cranium, our in-house ETL platform. Cranium allows us to write ETL jobs using SQL and Ruby, define dependencies between these jobs, and schedule their execution. Cranium supports ad-hoc reporting using Ruby and D3, but most data visualization happens in Chartio. We've found Chartio to be pretty intuitive for folks with limited SQL experience.

With these tools in place, everyone at from engineering to business development to community has no problem digging into the data to answer questions and discover insights.



Data visualization in Chartio

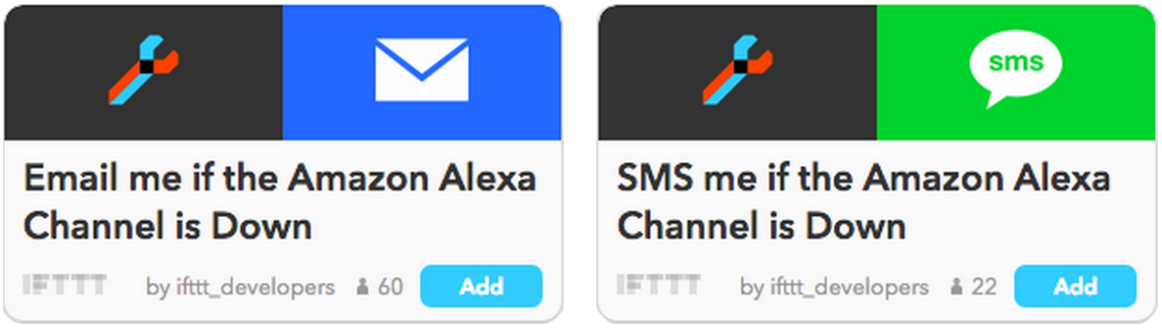
Machine Learning

We employ some sophisticated machine learning techniques to ensure that IFTTT users have a great experience. For Recipe recommendations and abuse detection, we use Apache Spark running on EC2 and use S3 as its data store. More on that in a future blog post.

Real-time Monitoring and Alerting

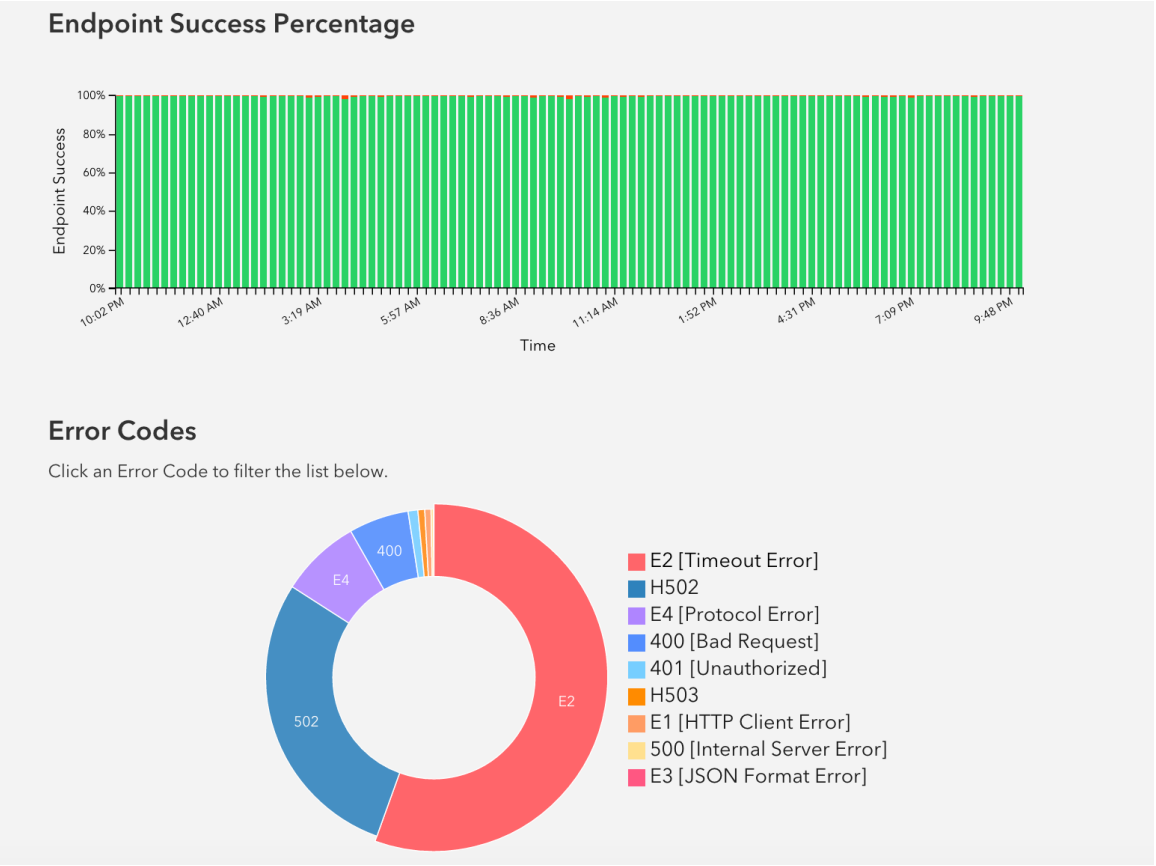
API events are stored in Elasticsearch for real-time monitoring and alerting. We use Kibana to visualize the performance of our worker processes, and the performance of partner APIs in real-time.

IFTTT partners have access to the Developer Channel, a special Channel that triggers when their API is having issues. They can create Recipes using the Developer Channel that notify them using the action Channel of their choice (SMS, Email, Slack, etc).



Partners can create Recipes for real-time alerts

Within the developer dashboard, partners can access real-time logs and visualizations of their Channel's health. These are powered by Elasticsearch as well. Additionally, developers are given powerful analytics to help them understand who is using their Channel and how.



Real-time Channel monitoring in the developer dashboard

Lessons

We've gained some great insights from developing our data infrastructure:

- Separation between producers and consumers through a data transport layer like Kafka is pure bliss, and makes the data pipeline much more resilient. For example, a few slow consumers won't impact the performance of the other consumers or producers.
- Start with a cluster from day one so that you can scale up easily, but make sure that you identify your bottleneck before blindly throwing nodes at a performance issue. For example, in Elasticsearch, if your shards are really large, adding more nodes may not help much for speeding up queries. You have to reduce the shard size to see improvement.
- In a complex architecture like this, it is critical to have the appropriate alerts in place to make sure everything is working fine. We use Sematext to monitor our Kafka cluster and consumers. We also use Pingdom for monitoring, and Pagerduty for alerts.
- In order to fully trust your data, it is important to have few automatic data verification steps in the flow. For example, we developed a service to compare the number of rows in a production table to the number of rows in the Redshift

table and send alerts if anything looks suspicious.

- Use date based folder structure (YYYY/MM/DD) to store event data in permanent storage (S3 in our case). Event data stored in this way is easy to process. For example if you want to read a particular day's data, you just need to read data from one directory.
- Similar to the above, create time based indexes (ex: hourly) in Elasticsearch. This way if you query Elasticsearch to find all API errors in the last hour, it can find the answer by looking at a single index, increasing efficiency.
- Rather than pushing individual events to Elasticsearch, push events in the batches (based on a time duration and/or number of events). This helps limit IO.
- Depending on the type of data and queries you are running, it is important to optimize number of nodes, number of shards, maximum size of each shard and replication factor in Elasticsearch.