

Quastor Archives

These are the full archives of the [Quastor Newsletter](#) until July of 2023.

Check out [the website](#) for more recent archives.

You can sign up to get the newsletter weekly [here](#).

Table of Contents

Quastor Archives	1
Table of Contents	2
Measuring Availability	9
Service Level Agreement	9
Availability	11
Latency	13
Other Metrics	14
Load Balancing Strategies	15
The Purpose of Load Balancers	15
Load Balancer vs. API Gateway	16
Types of Load Balancers	16
Load Balancing Algorithms	19
Scaling Relational Databases with Replicas and Sharding	21
Optimizations	21
Vertical Scaling	21
Adding Read Replicas	22
Sharding	23
Horizontal Partitioning Strategies	24
Backend Caching	28
Downsides of Caching	29
Implementing Caching	30
Cache Aside	30
Write Through	31
Cache Eviction	32
API Gateways	34
API Gateway use cases	35
API Gateway Lifecycle	36
History of API Gateways	39
API Gateways	40
Real World Uses	41
Zuul, Netflix's API Gateway	41
TAG, Tinder API Gateway	42
An Introduction to Compilers and LLVM	43
Introduction to Classical Compiler Design	44
LLVM's Implementation of the Three-Phase Design	46
How WhatsApp served 1 billion users with only 50 engineers.	50
Engineering Culture	50

Tech Stack	53
The Architecture of Uber's API gateway	56
How does the API Gateway work	56
How a request flows through the API gateway	57
Map Reduce Explained	60
History behind Map Reduce	60
Map and Reduce functions	61
How MapReduce Works	62
Building a basic Storage Engine	67
Log Structured Storage Engines	68
Robinhood's Tech Stack	72
Robinhood's Tech Stack	72
PaaS vs. DIY	72
How Robinhood Scaled	73
How Slack Designs APIs	74
Slack's API Design Principles	75
How Notion sharded their Postgres Database	77
When to Shard?	77
Application-Level vs. Managed	78
Shard Key	78
How many Shards?	79
Database Migration	79
Google File System	80
Goals of GFS	80
Design of GFS	81
GFS Mutations	83
GFS Interface	84
Scaling an API with Rate Limiters	85
Building a rate limiter in practice	87
GitHub's transition from Monolith to Microservices	88
History	88
Pros of a Monolith architecture	89
Pros of a Microservice architecture	89
How to break up the Monolith	90
Separating Data	90
Separating Services	91
Partitioning GitHub's Relational Database	93
Virtual Partitions	94
Moving Data without Downtime	96
LinkedIn's journey of scaling HDFS to 1 Exabyte	97
Replicating NameNodes	98

Java Tuning	101
Other Optimizations	102
Building a Static Analysis tool at Slack	104
How Facebook Encodes Videos	109
Facebook's Process for Encoding Videos	111
How Uber Migrated their Financial Transaction Database from DynamoDB to Docstore	114
Choosing Docstore	116
DynamoDB to Docstore Migration	117
An Introduction to Big Data Architectures	120
Components of a Big Data Architecture	120
Lambda Architecture	122
How Stripe uses Similarity Clustering to catch Fraud Rings	124
Merchant Fraud at Stripe	124
Using Similarity Clustering to Reduce Merchant Fraud	125
Switching from Heuristics-based to an ML model	125
Building the ML Model	126
Using gradient-boosted decision trees	126
Prediction Use	127
Evolving LinkedIn's Analytics Tech Stack	128
Data Migration	130
The New System	131
Etsy's Journey to TypeScript	133
Strategies for Adoption	133
Gradually Migrate to Strict TypeScript	134
Make sure Utilities and Tools have good TypeScript support	135
Educate and Onboard Engineers Team by Team	135
How Khan Academy rewrote their Backend	137
Brief Overview of Go	138
Monolith to Services	139
The Implementation	140
Final Results	142
Redesigning Etsy's Machine Learning Platform	143
The design of ML Platform V2	144
Outcomes	146
How Video Works	148
Playback	148
HLS	149
MP4 & WebM	150
Delivery	151
How Grab Processes Billions of Events in Real Time	153
The Architecture of Trident	154

Scalability	155
Scaling the Server Level	156
Scaling the Data Store Level	157
Lessons Learned from Implementing Payments in the DoorDash Android App	159
Why DoorDash migrated from Python to Kotlin	162
Summary	162
Clock Synchronization and NTP	166
Building Faster Indexing with Apache Kafka and Elasticsearch	169
DoorDash's Problem with Search Indexing	169
The New System	170
Incremental Indexing	171
Indexing Human Operator Changes	171
Indexing ETL data	172
Sending documents to Elasticsearch	172
Results	173
The Architecture of Databases	174
Observability at Twitter	178
The Legacy Logging System	178
Transitioning to Splunk	180
Challenges of running Splunk	182
Language Implementations Explained	184
Languages vs. Language Implementations	184
The Architecture of a Language Implementation	185
Front end	186
Middle End	187
Back End	188
Airbnb's Architecture	189
Monolith (2008 - 2017)	189
Microservices (2017 - 2020)	190
Micro + Macroservices (2020 -)	191
The Architecture of Apache Spark	193
History of MapReduce	193
How MapReduce Works	194
Issues with MapReduce	194
Creation of Apache Spark	195
Overview of Spark	196
Why is Spark Fast?	197
Architecture of Spark	198
Leader-Worker Architecture	198
Resilient Distributed Dataset	199
Directed Acyclic Graph	200

Architecture	204
Bloom Filters	206
Scaling an API with Rate Limiters	209
Building a rate limiter in practice	211
Serving Feature Data at Scale	212
Feature Definitions	213
Feature Data Ingestion	213
Feature Processing and Retrieval	214
Etsy's Journey to TypeScript	215
Strategies for Adoption	215
Gradually Migrate to Strict TypeScript	217
Make sure Utilities and Tools have good TypeScript support	217
Educate and Onboard Engineers Team by Team	218
Managing Infrastructure with Code at Shopify	219
Sharding Databases at Quora	222
MySQL at Quora	222
Splitting by Table	223
Splitting Individual Tables	224
Key Decisions around Sharding	225
How Quora Shards Tables	226
Video Delivery at Twitter with HTTP Live Streaming	229
Site Reliability Engineering at BlackRock	233
Architecture of the Telemetry Platform	233
Alerting Strategy	235
How Clubhouse Recommends Rooms	237
Complexities	239
Continuous Delivery at Airbnb	241
How BuzzFeed optimized their Frontend	245
Optimizations	248
Result	249
The Evolution of Benchling's Search Architecture	250
How the BBC uses Serverless	256
The BBC's Backend	257
Optimizing Performance	259
How Twitch does Chaos Engineering	262
How PayPal uses Graph Databases for Fraud Prevention	267
Properties	269
Graph Database	270
An Overview of PayPal's Graph Platform	271
Real Time Graph Database	272
Client Side Localization at Lyft	274

How Airbnb rebuilt their Payments System	279
Unified Entry Point	281
Improving Performance	282
Dropbox's Asynchronous Task Framework	285
ATF Architecture	287
The Architecture of Facebook's distributed Key Value store	290
Data Model	291
Architecture	291
Consistency	293
Challenges with Distributed Systems	295
Types of Distributed Systems	295
Complexity	296
Handling Failure Modes and Testing	297
Distributed Bugs are Often Latent	298
Distributed Bugs Spread Epidemically	298
How PayPal solved their Thundering Herd Problem	300
The Problem	301
Exponential Backoff	303
Jitter	304
How Twitch Processes Millions of Video Streams	306
Creating Intelligent	308
How Image Search works at Dropbox	310
Dropbox's Approach	311
Image Classification	311
Word Vectors	312
Production Architecture	312
How Instagram Suggests New Content	314
How Instagram does Candidate Generation	315
Cold Start Problem	316
How Instagram does Candidate Selection	316
How Snapchat Works	318
How Netflix Implemented Load Shedding	321
Define a Request Taxonomy	323
Load Shedding Algorithm	325
Validating Assumptions using Chaos Testing	326
The Architecture of Facebook's Distributed Message Queue	328
FOQs Use Cases	328
Building a Distributed Priority Queue	329
Enqueue	330
Dequeue	331
Ack/Nack	332

How Mixpanel Fixed their Load Balancing Problem	334
A Skew Problem	336
Cause for Skew	336
The Power of 2-Choices	337
How Pinterest Load Tests Their Database	338
Setting up the Testing Environment	340
Evaluating Query Load	342
Handling Increase in Data Ingestion	343
Handling Increase in Data Volume	343
Results	344
How Facebook Transfers Exabytes of Data Across Their Data Centers Globally	345
Hierarchical caching	347
Bittorrent	347
Owl	348
Results	350
How Dropbox maintains 3 Nines of Availability	352
Detection	353
Diagnosis	353
Recovery	354

Measuring Availability

When you're building a system, an incredibly important consideration you'll have to deal with is availability.

You'll have to think about

- What availability guarantees do you provide to your end users
- What availability guarantees do dependencies you're using provide to you

These availability goals will affect how you design your system and what tradeoffs you make in terms of redundancy, autoscaling policies, message queue guarantees, and much more.

Service Level Agreement

Availability guarantees are conveyed through a Service Level Agreement (SLAs).

Services that you use will provide one to you and you might have to give one to your end users (either external users or other developers in your company who rely on your API).

Here's some examples of SLAs

- [Google Cloud Compute Engine SLA](#)
- [AWS RDS Service Level Agreement](#)
- [Azure Kubernetes Service SLA](#)

These SLAs provide monthly guarantees in terms of Nines. We'll discuss this shortly. If they don't meet their availability agreements, then they'll refund a portion of the bill.

Service Level Agreements are composed of multiple Service Level Objectives (SLOs). An SLO is a specific target level objective for the reliability of your service.

Examples of possible SLOs are

- be available 99.9% of the time, with a maximum allowable downtime of ~40 minutes per month.
- respond to requests within 100 milliseconds on average, with no more than 1% of requests taking longer than 200 milliseconds to complete (P99 latency).
- handle 1,500 requests per second during peak periods, with a maximum allowable response time of 200 milliseconds for 99% of requests.

SLOs are based on Service Level Indicators (SLI), which are specific measures (indicators) of how the service is performing.

The SLIs you set will depend on the service that you're measuring. For example, you might not care about the response latency for a batch logging system that collects a bunch of logging data and transforms it. In that scenario, you might care more about the *recovery point objective* (maximum amount of data that can be lost during the recovery from a disaster) and say that no more than 12 hours of logging data can be lost in the event of a failure.

The Google SRE Workbook has a [great table](#) of the types of SLIs you'll want depending on the type of service you're measuring.

Availability

Every service will need a measure of availability. However, the exact definition will depend on the service.

You might define availability using the SLO of "successfully responds to requests within 100 milliseconds". As long as the service meets that SLO, it'll be considered available.

Availability is measured as a proportion, where it's *time spent available / total time*. You have ~720 hours in a month and if your service is available for 715 of those hours then your availability is 99.3%.

It is usually conveyed in nines, where the nine represents how many 9s are in the proportion.

If your service is available 92% of the time, then that's 1 nine. 99% is two nines. 99.9% is three nines. 99.99% is four nines, and so on. The gold standard is 5 Nines of availability, or available at least 99.999% of the time.

When you talk about availability, you also need to talk about the unit of time that you're measuring availability in. You can measure your availability weekly, monthly, yearly, etc.

If you're measuring it weekly, then you give an availability score for the week and that score resets every week.

So, if you measure downtime monthly, then you must have *less than 40 minutes* of downtime to have 3 Nines.

Availability	Downtime Per Week	Downtime Per Month	Downtime Per Year
90%+ (1 Nine)	16 hours	72 hours	36 days
99%+ (2 Nines)	1.5 hours	7.2 hours	3 days
99.9%+ (3 Nines)	10 minutes	40 minutes	8 hours
99.99%+ (4 Nines)	1 minute	4 minutes	50 minutes
99.999%+ (5 Nines)	6 seconds	25 seconds	5 minutes

So, let's say it's the week of May 1rst and you have 5 minutes of downtime that week. For the week of May 7th, your service has 30 seconds of downtime. Then you'll have 3 Nines of availability for the week of May 1rst and 4 Nines of availability for the week of May 7th.

However, if you were measuring availability monthly, then the moment you had that 5 minutes of downtime in the week of May 1rst, your availability for the month would've been at most 3 Nines. Having 4 Nines means less than 4 minutes, 21 seconds of downtime, so that would've been impossible for the month.

[Here's](#) a calculator that shows daily, weekly, monthly and yearly availability calculations for the different Nines.

Most services will measure availability monthly. At the end of every month, the availability proportion will reset. You can read more about choosing an appropriate time window [here](#) in the Google SRE workbook.

Latency

An important way of measuring the availability of your system is with the latency. You'll frequently see SLOs where the system has to respond within a certain amount of time in order to be considered available.

It's important to distinguish between the latency of a successful request vs. an unsuccessful one. For example, if your server has some configuration error (or some other error), it might immediately respond to any HTTP request with a **500**. Computing these latencies with your successful responses will throw off the calculation.

There's different ways of measuring latency, but you'll commonly see two ways

- Averages - Take the mean or median of the response times. If you're using the mean, then tail latencies (extremely long response times due to network congestion, errors, etc.) can throw off the calculation.
- Percentiles - You'll frequently see this as *P99* or *P95* latency (99th percentile latency or 95th percentile latency). If you have a P99 latency of 200 ms, then 99% of your responses are sent back *within* 200 ms.

Latency will typically go hand-in-hand with throughput, where throughput measures the number of requests your system can process in a certain interval of time (usually measured in requests per second). As the requests per second goes up, the latency will go up as well. If you have a sudden spike in requests per second, users will experience a spike in latency until your backend's autoscaling kicks in and you get more machines added to the server pool.

You can use load testing tools like [JMeter](#), [Gatling](#) and more to put your backend under heavy stress and see how the average/percentile latencies change.

The high percentile latencies (have a latency that is slower than 99.9% or 99.99% of all responses) might also be important to measure, depending on the application. These latencies can be caused by network congestion, garbage collection pauses, packet loss, contention, and more.

To track tail latencies, you'll commonly see **histograms** and **heat maps** utilized.

Other Metrics

There's an infinite number of other metrics you can track, depending on what your use case is. Your customer requirements will dictate this.

Some other examples of commonly tracked SLOs are MTTR, MTBM and RPO.

MTTR - Mean Time to Recovery measures the average time it takes to repair a failed system. Given that the system is down, how long does it take to become operational again? Reducing the MTTR is crucial to improving availability.

MTBM - Mean Time Between Maintenance measures the average time between maintenance activities on your system. Systems may have scheduled downtime (or degraded performance) for maintenance activities, so MTBM measures how often this happens.

RPO - Recovery Point Objective measures the maximum amount of data that a company can lose in the event of a disaster. It's usually measured in time and it represents the point in time when the data must be restored in order to minimize business impact. If a company has an RPO of 2 hours, then that means that the company can tolerate the loss of data up to 2 hours old in the event of a disaster. RPO goes hand-in-hand with MTTR, as a short RPO means that the MTTR must also be very short. If the company can't tolerate a significant loss of data when the system goes down, then the Site Reliability Engineers must be able to bring the system back up ASAP.

Load Balancing Strategies

The Purpose of Load Balancers

As your backend gets more traffic, you'll eventually reach a point where vertically scaling your web server (upgrading your hardware) becomes too costly. You'll have to scale horizontally and create a server pool of multiple machines that are handling incoming requests.

A load balancer sits in front of that server pool and directs incoming requests to the servers in the pool. If one of the web servers goes down, the load balancer will stop sending it traffic. If another web server is added to the pool, the load balancer will start sending it requests.

Load balancers can also handle other tasks like caching responses, handling session persistence (send requests from the same client to the same web server), rate limiting and more.

Typically, the web servers are hidden in a private subnet (keeping them secure) and users connect to the public IP of the load balancer. The load balancer is the “front door” to the backend.

Load Balancer vs. API Gateway

When you're using a services oriented architecture, you'll have an API gateway that directs requests to the corresponding backend service. The API Gateway will also provide other features like rate limiting, circuit breakers, monitoring, authentication and more. The API Gateway can act as the front door for your application instead of a load balancer.

API Gateways can replace what a load balancer would provide, but it'll usually be cheaper to use a load balancer if you're not using the extra functionality provided by the API Gateway.

Here's a great [blog post](#) that gives a detailed comparison of AWS API Gateway vs. AWS Application Load Balancer.

Types of Load Balancers

When you're adding a load balancer, there are two main types you can use: layer 4 load balancers and layer 7 load balancers.

This is based on the [OSI Model](#), where layer 4 is the transport layer and layer 7 is the application layer.

The main transport layer protocols are TCP and UDP, so a L4 load balancer will make routing decisions based on the packet headers for those protocols: the IP address and the port. You'll frequently see the terms “4-tuple” or “5-tuple” hash when looking at L4 load balancers.

This hash is based on the "5-Tuple" [concept in TCP/UDP](#).

- Source IP
- Source Port
- Destination IP
- Destination Port
- Protocol Type

With a 5 tuple hash, you would use all 5 of those to create a hash and then use that hash to determine which server to route the request to. A 4 tuple hash would use 4 of those factors.

With Layer 7 load balancers, they operate on the application layer so they have access to the [HTTP headers](#). They can read data like the URL, cookies, content type and other headers. An L7 load balancer can consider all of these things when making routing decisions.

Popular load balancers like HAProxy and Nginx can be configured to run in layer 4 or layer 7. AWS Elastic Load Balancing service provides Application Load Balancer (ALB) and Network Load Balancer (NLB) where ALB is layer 7 and NLB is layer 4 (there's also Classic Load Balancer which allows both).

The main benefit of an L4 load balancer is that it's quite simple. It's just using the IP address and port to make its decision and so it can handle a very high rate of requests per second. The downside is that it has no ability to make smarter load balancing decisions. Doing things like caching requests is also not possible.

On the other hand, layer 7 load balancers can be a lot smarter and forward requests based on rules set up around the HTTP headers and the URL parameters. Additionally, you can do things like cache responses for GET requests for a certain URL to reduce load on your web servers.

The downside of L7 load balancers is that they can be more complex and computationally expensive to run. However, CPU and memory are now sufficiently fast and cheap enough that the performance advantage for L4 load balancers has become pretty negligible in most situations.

Therefore, most general purpose load balancers operate at layer 7. However, you'll also see companies use both L4 and L7 load balancers, where the L4 load balancers are placed before the L7 load balancers.

Facebook has a setup like this where they use shiv (a L4 load balancer) in front of proxygen (a L7 load balancer). You can see a talk about this set up [here](#).

Load Balancing Algorithms

Round Robin - This is usually the default method chosen for load balancing where web servers are selected in [round robin](#) order: you assign requests one by one to each web server and then cycle back to the first server after going through the list. Many load balancers will also allow you to do [weighted round robin](#), where you can assign each server weights and assign work based on the server weight (a more powerful machine gets a higher weight).

An issue with Round Robin scheduling comes when the incoming requests vary in processing time. Round robin scheduling doesn't consider how much computational time is needed to process a request, it just sends it to the next server in the queue. If a server is next in the queue but it's stuck processing a time-consuming request, Round Robin will still send it another job anyway. This can lead to a work skew where some of the machines in the pool are at a far higher utilization than others.

Least Connections (Least Outstanding Requests) - With this strategy, you look at the number of active connections/requests a web server has and also look at server weights (based on how powerful the server's hardware is). Taking these two into consideration, you send your request to the server with the least active connections / outstanding requests. This helps alleviate the work skew issue that can come with Round Robin.

Hashing - In some scenarios, you'll want certain requests to always go to the same server in the server pool. You might want all GET requests for a certain URL to go to a certain server in the pool or you might want all the requests from the same client to always go to the same server ([session persistence](#)). Hashing is a good solution for this.

You can define a key (like request URL or client IP address) and then the load balancer will use a hash function to determine which server to send the request to. Requests with the same key will always go to the same server, assuming the number of servers is constant.

Consistent Hashing - The issue with the hashing approach mentioned above is that adding/removing servers to the server pool will mess up the hashing scheme. Anytime a

server is added, each request will get hashed to a new server. Consistent hashing is a strategy that's meant to minimize the number of requests that have to be sent to a new server when the server pool size is changed. [Here's a great video](#) that explains why consistent hashing is necessary and how it works.

There are different consistent hashing algorithms that you can use and the most common one is [Ring hash](#). [Maglev](#) is another consistent hashing algorithm that was developed by Google in 2016 and has been serving Google's traffic since 2008.

Scaling Relational Databases with Replicas and Sharding

A vital part of scaling a web application is making sure your database can deal with the increase in read/write load. For relational databases, the playbook revolves around vertical scaling with getting a more powerful machine and horizontal scaling with replicas and sharding.

For most applications, the read load will be far greater than the write load, so you'll have to deal with scaling reads first. This is done through adding read replicas to the database so the original database can just handle write requests and the database replicas will handle reads.

Eventually, the master won't be able to handle all the write pressure, so a popular way of dealing with that is through sharding. You split up your data into multiple databases, so you can dedicate a machine to handling each chunk of data.

We'll go through these strategies below.

Optimizations

Before you think about adding replicas or sharding, you should first explore all your options around database optimizations.

Using the right indexes on tables, batching writes, analyzing the execution plan for your most expensive SQL queries, etc. should all be done first.

Vertical Scaling

After exploring possible optimizations, vertical scaling will likely give you the biggest bang for your buck. Getting a beefier CPU, more RAM, faster disk etc. should be considered before scaling with replicas and sharding.

Adding Read Replicas

Most web applications have far more read load than write load, so reducing that pressure by adding read replicas can have a huge impact on your database's scalability.

Your original database becomes the leader and you add additional machines that function as replicas/followers. Database reads are completed by the follower nodes while writes are executed on the master node. After being executed on the master, writes are copied over asynchronously to follower nodes.

Due to replication lag between the master and follower nodes, you'll be sacrificing consistency and have to deal with some stale reads.

One way of fixing this is to implement different read modes with *strongly consistent* reads and *eventually consistent* reads.

Eventually consistent reads go to a follower node whereas strongly consistent reads will be handled by the leader node. This lets you reduce read load on the leader database while also having the ability to run strongly consistent read queries.

Another way to make reads strongly consistent is to wait until all the replicas have gotten the write before you acknowledge it as being completed to the user.

[Here's](#) a great blog post from Box on how they dealt with consistency issues with read replicas.

However, adding read replicas will not help to scale the write load in any way. For that, you'll have to shard your database.

Sharding

With sharding, you're taking your database and splitting up the data to store on multiple databases. This lets you use multiple machines to store the same data, which makes it easier to deal with write and read load (more machines to handle the traffic).

Some of the downsides are

- Adding complexity to your backend - Sharding is frequently implemented in the application logic, so this means more complexity for your codebase. Plus, you have all the standard stuff that makes distributed systems really hard (network failures, synchronization issues, outages, etc.)
- Sacrificing Consistency - This is similar to what we discussed for adding read replicas. Having multiple machines means data needs to be replicated asynchronously or writes have to be done synchronously.
- Increased latency for certain reads - if a read needs data from multiple shards, it will have to perform reads from multiple databases and then join the data together.

You can break down your data using *horizontal partitioning* where you maintain the table schema and columns, but have each shard contain a number of the rows of your table.

Vertical Partitioning is where you split up your table based on the columns of your table.

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKO	BLUE
2	O.V.	WRIGHT	GREEN
3	SELDI	BAŞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

Vertical Partitions

VP1			VP2	
CUSTOMER ID	FIRST NAME	LAST NAME	CUSTOMER ID	FAVORITE COLOR
1	TAEKO	OHNUKO	1	BLUE
2	O.V.	WRIGHT	2	GREEN
3	SELDI	BAŞCAN	3	PURPLE
4	JIM	PEPPER	4	AUBERGINE

Horizontal Partitions

HP1				HP2			
CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR	CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKO	BLUE	3	SELDI	BAŞCAN	PURPLE
2	O.V.	WRIGHT	GREEN	4	JIM	PEPPER	AUBERGINE

You can implement sharding at the Application layer (where you'll have code that implements the partitioning in your application logic) or it can be done by the database management system.

You can [read about how Notion](#) sharded Postgres by implementing their own partitioning scheme in their application logic.

In terms of potential third party solutions, [Vitess](#) is an open source solution for sharding MySQL that was developed at YouTube. [Citus](#) is a similar open source tool for sharding Postgres.

You can read about how GitHub used Vitess to shard MySQL [here](#).

Horizontal Partitioning Strategies

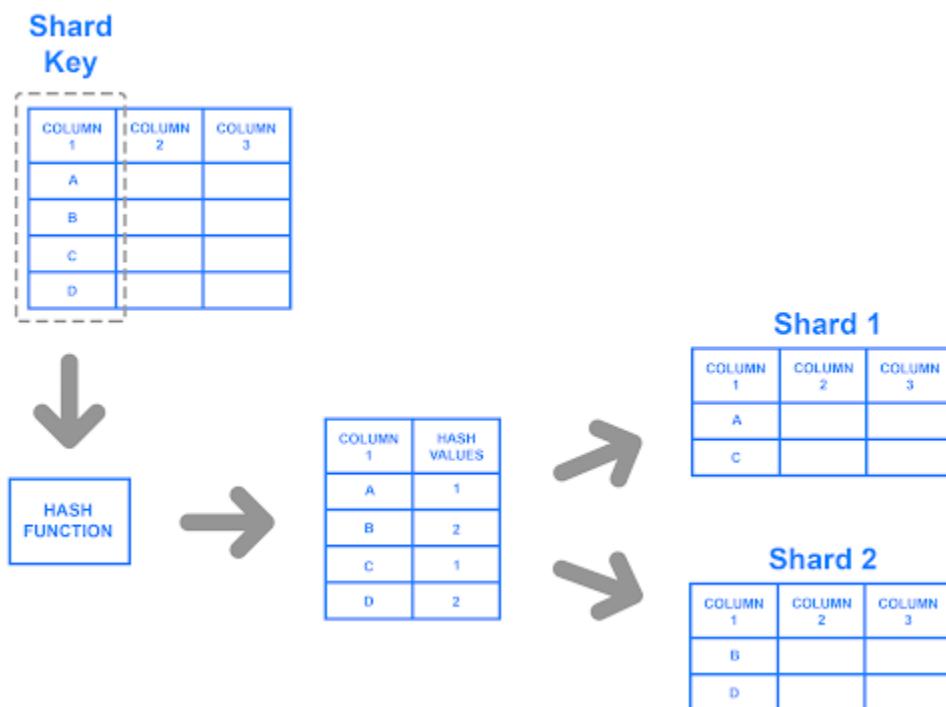
With horizontal partitioning, you're taking one (or multiple) of the fields in your table and making that your *shard key*.

You'll use the shard key's value to determine which database shard a row goes into. The shard key should be selected so that each database shard gets an equal number of read/write queries. You want to minimize the number of hot database shards you have (shards that get more load than the others).

Some ways of using the shard key to partition are hash based sharding, range based sharding and lookup table based sharding.

Hash Based Sharding

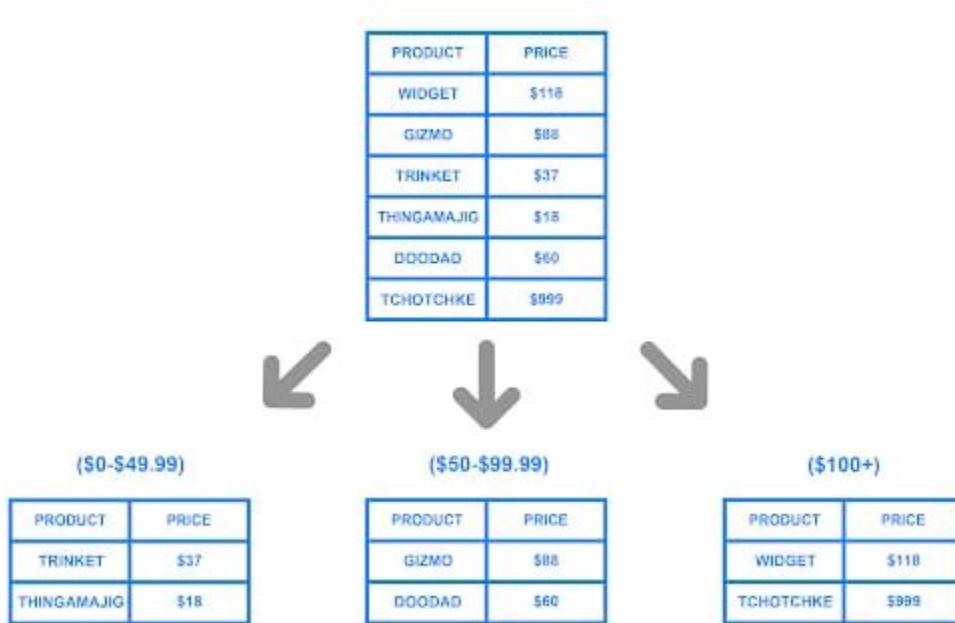
You run a hash function on the value of the shard key and then categorize it based on the hashed value. A good hash function will satisfy the [uniformity property](#), so the outputs will be mapped evenly over the output range. This should help mitigate hot/cold shards.



Range Based Sharding

With this strategy, you'll split up your partition key into ranges and then divide the rows into shards based on that.

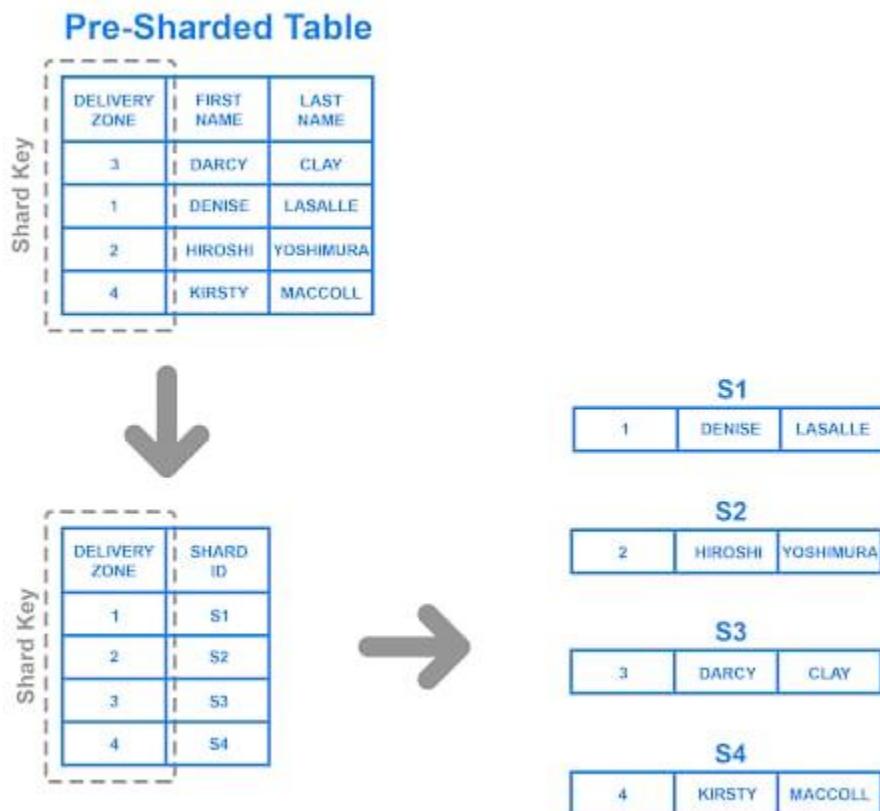
Examples of ranges could be location (each country is a shard), date (each month is a shard), price (every \$100 increment is a shard), etc.



Lookup Table Based Sharding

A third way of implementing sharding is by using a lookup table (or hash table).

Like we did in key-based sharding, you'll set one column of your data as the shard key. Then, you can randomly assign rows to different shards and keep track of which shard contains which row with a lookup table.



Backend Caching

Caching is a crucial part of large scale, performant system design. For many web applications, the database workload will be read intensive (users will send far more database read requests than writes) so finding a way to reduce the read load on your database can be very useful.

If the data being read doesn't change often, then adding a caching layer can significantly reduce the latency users experience while also reducing the load on your database. The reduction in read requests frees up your database for writes (and reads that were missed by the cache).

The cache tier is a data store that temporarily stores frequently accessed data. It's not meant for permanent storage and typically you'll only store a small subset of your data in this layer.

When a client requests data, the backend will first check the caching tier to see if the data is cached. If it is, then the backend can retrieve the data from cache and serve it to the client. This is a cache hit. If the data isn't in the cache, then the backend will have to query the database. This is a cache miss. Depending on how the cache is set up, the backend might write that data to the cache to avoid future cache misses.

A couple examples of popular data stores for caching tiers are Redis, Memcached, Couchbase and Hazelcast. Redis and Memcached are the most popular and they're offered as options with cloud cache services like AWS's [ElastiCache](#) and Google Cloud's [Memorystore](#).

Redis and Memcached are in-memory, key-value data stores, so they can serve reads with a lower latency than disk-based data stores like Postgres. They're in-memory data stores, so RAM is used as the primary method of storing and serving data while the disk is used for backups and logging. This translates to speed improvements as memory is much faster than disk.

Downsides of Caching

If implemented poorly, caching can result in increased latency to clients and add unnecessary load to your backend. Everytime there's a cache miss, then the backend has just wasted time making a request to the caching tier.

If you have a high cache miss rate, then that means the caching tier is adding more latency than it's reducing and you'd be faster off by removing it. We'll talk about strategies to minimize the cache miss rate.

Another downside of adding a cache tier is dealing with stale data. If the data you're caching is static, then this isn't an issue but you'll frequently want to cache data that is being changed. You'll have to have a strategy for [cache invalidation](#) to minimize the amount of stale data you're sending to clients. We'll talk about this below.

Implementing Caching

There are several ways of implementing caching in your application. We'll go through a few of the main ones.

Cache Aside

The most popular method is a Cache Aside strategy.

Here's the steps

1. Client requests data
2. The server checks the caching tier. If there's a cache hit, then the data is immediately served.
3. If there's a cache miss, then the server checks the database and returns the data.
4. The server writes the data to the cache.

Here, your cache is being loaded *lazily*, as data is only being cached after it's been requested. You usually can't store your entire dataset in cache, so lazy loading the cache is a good way to make sure the most frequently read data is cached.

However, this also means that the first time data is requested will always result in a cache miss. Developers solve this by cache warming, where you load data into the cache manually.

In order to prevent stale data, you'll also give a Time-To-Live (TTL) whenever you cache an item. When the TTL expires, then that data is removed from the cache. Setting a very low TTL will reduce the amount of stale data but also result in a higher number of cache misses. You can read more about this tradeoff in this [AWS Whitepaper](#).

An alternative caching method that minimizes stale data is the Write-Through cache.

Write Through

A Write Through cache can be viewed as an *eager loading* approach. Whenever there's a change to the data, that change is reflected in the cache.

This helps solve the data consistency issues (avoid stale data) and it also prevents cache misses for when data is requested the first time.

Here's the steps.

1. Client writes/modifies data.
2. Backend writes the change to *both* the database and also to the cache. You can also do this step asynchronously, where the change is written to the cache and then the database is updated after a delay (a few seconds, minutes, etc.). This is known as a Write Behind cache.
3. Clients can request data and the backend will try to serve it from the cache.

A Write Through strategy is often combined with a Read Through so that changes are propagated in the cache (Write Through) but missed cache reads are also written to the cache (Read Through).

You can read about more caching patterns in the Oracle Coherence [Documentation](#) (Coherence is a Java-based distributed cache).

Cache Eviction

The amount of storage you have available in your caching tier is usually a lot smaller than what you have available for your database.

Therefore, you'll eventually reach a situation where your cache is full and you can't add any new data to it.

To solve this, you'll need a [cache replacement](#) policy. The ideal cache replacement policy will remove cold data (data that is not being read frequently) and replace it with hot data (data that is being read frequently).

There are many different possible cache replacement policies.

A few categories are

- Queue-Based - Use a FIFO queue and evict data based on the order in which it was added regardless of how frequently/recently it was accessed.
- Recency-Based - Discard data based on how recently it was accessed. This requires you to keep track of when each piece of data in your cache was last read. The Least Recently Used (LRU) policy is where you evict the data that was read least recently.
- Frequency-Based - Discard data based on how many times it was accessed. You keep track of how many times each piece of data in your cache is being read. The Least Frequently Used (LFU) policy is where you evict data that was read the least.

The type of cache eviction policy you use depends on your use case. Picking the optimal eviction policy can massively improve your cache hit rate.

Here's some helpful links on Caching Strategies. I referred to these sources while writing this article.

- [Caching patterns - AWS Whitepaper](#)
- [Cache-Aside pattern - Azure Architecture Center](#)
- [Read-Through, Write-Through, Write-Behind, and Refresh-Ahead Caching](#)
- [Cache replacement policies - Wikipedia](#)

API Gateways

Within your backend you'll have many different machines for the various services, databases, caches, load balancers, etc. These machines are all connected through an internal network; your [VPC](#).

Exposing any of your internal machines to the public internet is not a good idea. Anything that is exposed will need rate limiting and protections around DDoS. It's also a security risk; if hackers can see what software your servers are running, then they can look up any known vulnerabilities/exploits and attack you with them.

Instead, you'll want to have all your internal machines inside of a VPC and expose them through a single entrypoint: a reverse proxy. The reverse proxy is the only thing exposed to the public internet and it will have security protections against things like DDoS attacks.

Previously, the most common type of reverse proxy was a load balancer. When someone sends requests to your backend, they go to the load balancer as the entrypoint.

We talked in depth about load balancers in a previous Quastor Pro article [here](#).

With the rise of microservice architectures, API gateways have become significantly more popular as the entrypoint to your backend. They handle tasks like load balancing, but they can also do much more.

API Gateway use cases

As mentioned, API gateways serve as the entrypoint to your backend. When a client device sends a request to your backend, they're communicating with your API gateway,

The API Gateway will handle tasks like

- Load Balancing - Balance the requests between the different servers in your server pool. API gateways function in a similar way to an L7 load balancer.
- Caching - The gateway can cache the response to specific endpoints to reduce the number of calls made to your backend services. You can see how caching works with Amazon API gateway [here](#).
- Rate Limiting - Misconfigured/malicious clients can spam your backend with traffic, so it's important to block any clients that do this with a [429 response](#). This can be implemented within your API gateway so it has minimal impact on backend services. [Amazon API Gateway](#) does this with the token bucket rate limiting algorithm.
- Authentication and Authorization - Many API gateways will also authenticate the requests that come in and make sure the client is authorized to access the resources they're asking for.
- Service Discovery - When you use a microservice architecture, you'll have hundreds/thousands of backend services. The API gateway will keep track of which endpoints relate to which services and make the backend calls to the appropriate services.
- Protocol Translation - The client might be sending you requests with HTTP but your backend uses gRPC to communicate. The API gateway will handle this translation where it takes the HTTP request and converts it to gRPC. Then, it takes the backend service's response and creates the HTTP response to send back to the client.

- Monitoring, Logging and Analytics - The API Gateway will also handle monitoring so you can have detailed logs of all the traffic that hit your backend and all the responses that were sent back.

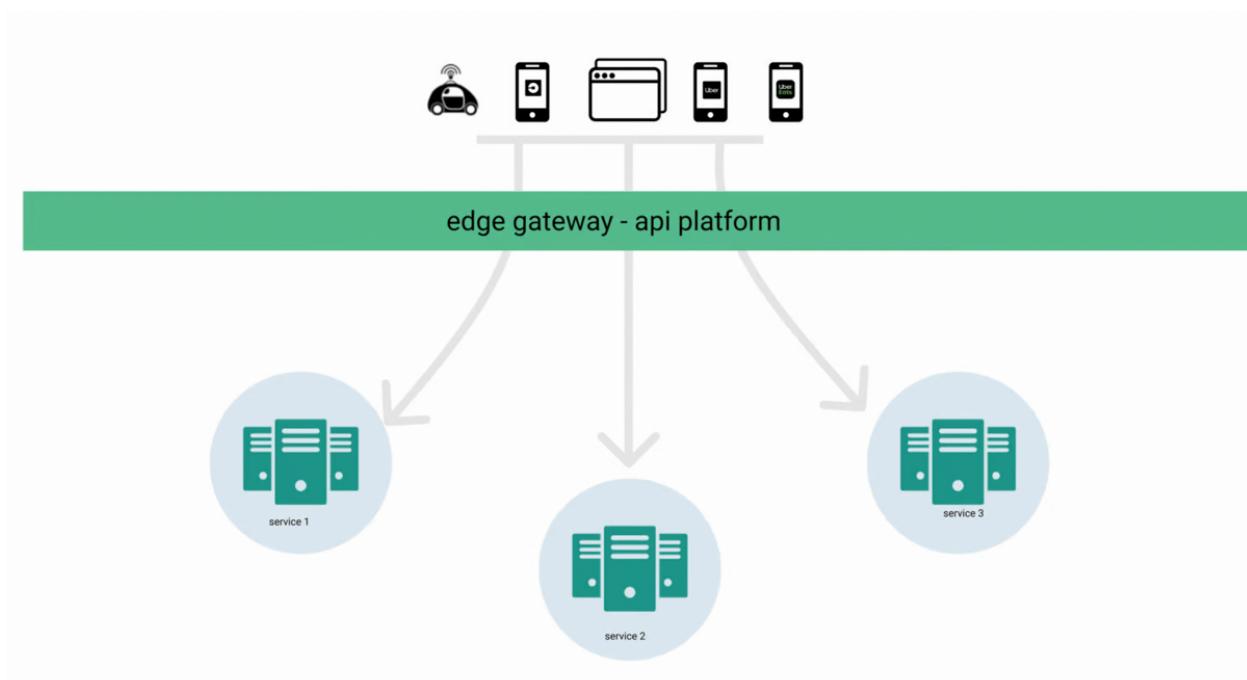
API gateways are primarily used with microservices architectures. You can use an API gateway with a monolith, but it's usually not necessary. The features around authentication/authorization and protocol translation are typically already handled by the monolith. You also don't need service discovery as there's just a single service.

Therefore, with monoliths, teams will usually just use load balancers as the entrypoint.

However, you'll also see engineering teams (with services-oriented architectures) use both. They'll have load balancers serve as the entry point to the application. Then, an API gateway will sit between the load balancers and all microservices in the backend.

API Gateway Lifecycle

Uber wrote a great [blog post](#) delving into how their API gateway works. We'll use this to illustrate the request lifecycle.



Uber developed their own API gateway to handle data payloads across multiple protocols and client device types. They use Nginx as the entrypoint to the backend, which handles load balancing and decryption. From there, the requests go to the API gateway.

The Gateway's lifecycle consists of several steps.

1. Protocol Manager - This layer contains a deserializer and serializer for all of the protocols supported by the gateway. The gateway has to deal with different types of protocol payloads like JSON, Thrift, Protobuf, etc. so this layer handles translation
2. Middleware - This layer contains different middleware functions for things like authentication, authorization, rate limiting, logging, monitoring, etc. Engineers can write the code for new middleware functions and add them to the API gateway through a UI
3. Endpoint Handler - This layer validates the request and transforms the request object into an object that backend services can understand.
4. Requests to Backend Services - This part of the backend is responsible for making the actual calls to backend microservices. It handles service discovery to figure out which microservices to call and also takes care of things like circuit breaking (so a microservice doesn't get overloaded) and error handling, timeouts, retries and more.

This describes how the API gateway takes in a request. To send a response back to the client, the request goes through the same components in reverse order.

Here's the lifecycle of sending a request back to the client.

1. Requests to backend services - This service has already made requests to various backend microservices and now it's received responses. This part of the gateway aggregates the responses and sends it to the endpoint handler.

2. Endpoint Handler - This converts the back-end service responses to the endpoint response schema. It performs transforms on the response objects to do this. It also performs validation on the endpoint response based on the schema.
3. Middleware - Uber has middleware functions that work on the response objects to handle things like logging and monitoring. These functions might also add things like HTTP headers to the response object.
4. Payload Manager - This transforms the backend response to the relevant payload protocol like JSON, Thrift, Protobuf, etc. Then it sends it back to the load balancer, who sends it to the client.

History of API Gateways

Previously, companies used monolithic architectures so they didn't need the feature set of an API gateway. Instead, they relied on load balancers to route requests to the server pool, where each machine was running an instance of the monolith.

F5 was founded in 1996, HAProxy in 2001 and NGINX in 2002. These companies all produced network/application load balancers to direct traffic to the servers in the pool.

Large tech companies like Amazon, Netflix, eBay and more began to adopt service oriented architectures. These companies (Netflix especially) began to publicly champion microservices and the term became commonly used by the early 2010s.

In mid-2013, Netflix released their JVM-based API gateway: Zuul. It has a ton of cool features like allowing Groovy scripts to be injected at runtime to add middleware and dynamically modify behavior. You could write middleware to handle things like rate limiting, load shedding, authentication, monitoring, release engineering (canary releases or A/B testing) and more.

In the mid to late 2010s, you had the adoption of containers and container management tools like Kubernetes. This led to the concept of a service mesh architecture with projects like Envoy by Lyft. These tools facilitate service discovery so developers can focus on writing the code for their microservice and let the service mesh tool handle inter-service communication. Companies like Snapchat use Envoy for their service mesh and also have it [serve as the API gateway](#).

API Gateways

Here are a couple popular options for API gateways

- Amazon API Gateway (plus managed solutions from other cloud providers)
- Kong Gateway
- Tyk Gateway
- Ambassador
- Nginx Plus
- Envoy

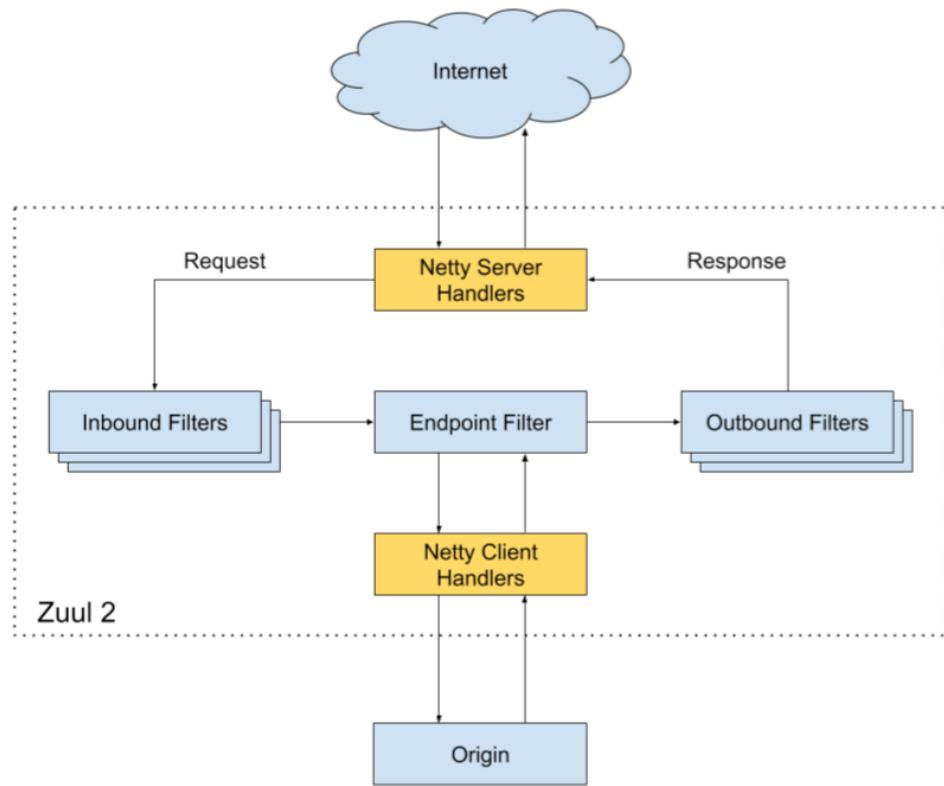
And more. Kong, Tyk, Ambassador and Envoy are open source.

Real World Uses

Zuul, Netflix's API Gateway

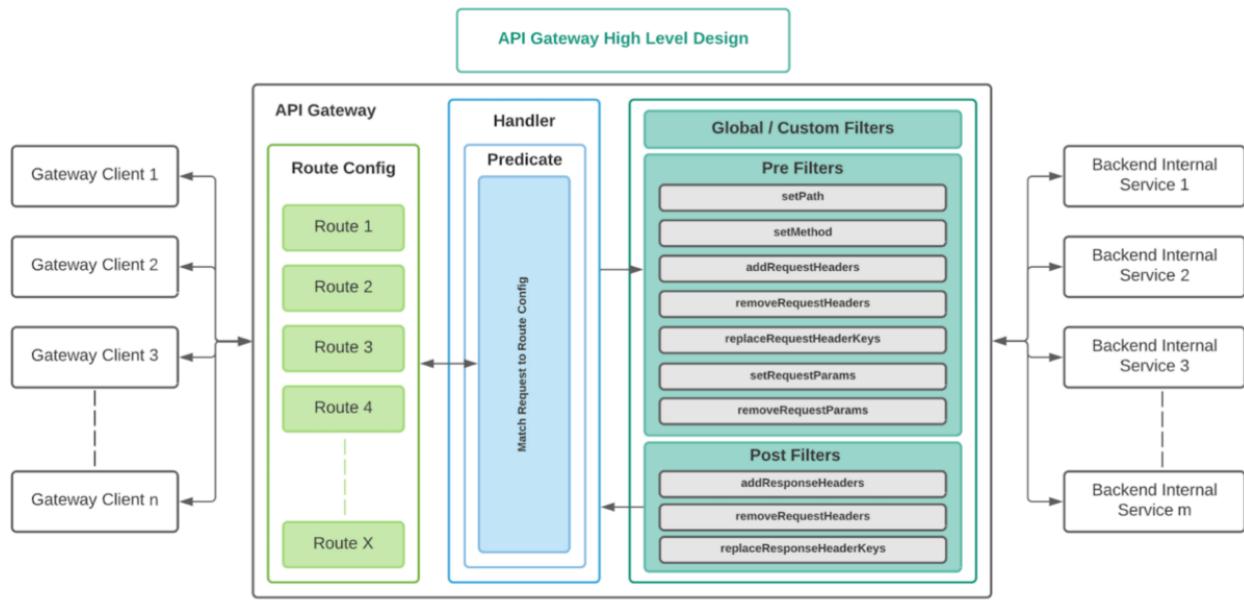
Netflix uses Zuul, which is built with Java. It handles tasks like load balancing, routing, monitoring, authentication, payload transformation, etc. They also built in features to increase reliability like load shedding and stress testing (to stress test backend microservices). You can read about how Zuul does Service Discovery for Netflix's thousands of microservices [here](#). The gateway also handles errors in the backend by categorizing the error message and running retries.

The [wiki](#) gives a great overview of how Netflix uses Zuul and the other tooling they built as complements to the gateway.



TAG, Tinder API Gateway

Tinder has more than 500 microservices that communicate with each other using a service mesh. They built TAG on top of [Spring Cloud Gateway](#), an API gateway that's part of the Java Spring ecosystem. The API gateway handles tasks like load balancing, transforming requests/responses, HTTP to gRPC conversion and more.



An Introduction to Compilers and LLVM

LLVM is an insanely cool set of low-level technologies (assemblers, compilers, debuggers) that are *language-agnostic*. They're used for C, C++, Ruby, C#, Scala, Swift, Haskell, and a ton of other languages.

This [post](#) goes through an introduction to compiler design, the motivations behind LLVM, the design of LLVM and some extremely useful features that LLVM provides.

We'll be giving a summary below.

Back in 2000, open source programming language implementations (interpreters and compilers) were designed as special purpose tools and were monolithic executables. It would've been difficult to reuse the parser from a static compiler to do static analysis or refactoring.

Additionally, programming language implementations usually provided *either* a traditional static compiler or a runtime compiler in the form of an interpreter or JIT compiler.

It was very uncommon to see a language implementation that had supported both, and if there was then there was very little sharing of code.

The LLVM project helped change this.

Introduction to Classical Compiler Design

The most popular design for traditional static compilers is the three-phase design where the major components are the front end, the optimizer, and the back end.



Front End

The front end parses the source code (checking it for errors) and builds a language-specific Abstract Syntax Tree.

The AST is optionally converted to a new representation for optimization (this may be a common code representation, where the code is the same regardless of the input source code's language).

Optimizer

The optimizer runs a series of *optimizing transformations* to the code to improve the code's running time, memory footprint, storage size, etc.

This is more or less *independent* of the input source code language and the target language

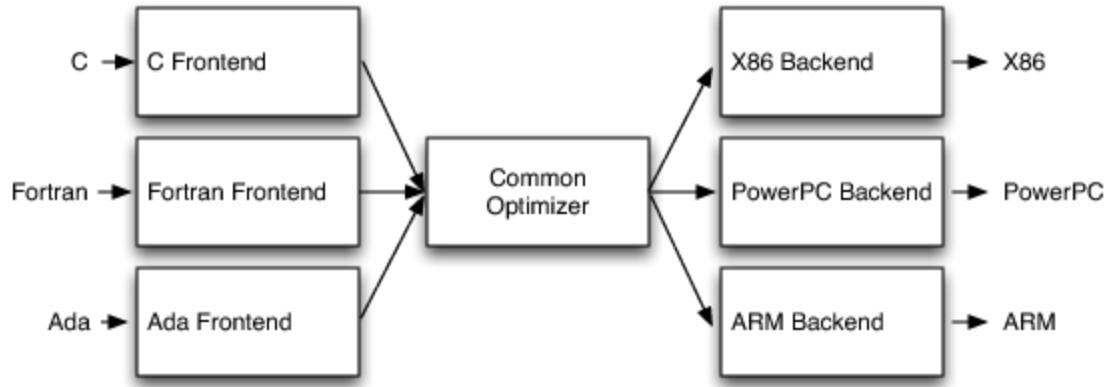
Back End

The back end maps the optimized code onto the target instruction set.

It's responsible for generating fast code that takes advantage of the specific features of the supported architecture.

Common parts of the back end include instruction selection, register allocation, and instruction scheduling.

The most important part of this design is that a compiler can be easily adapted to support multiple source languages or target architectures.



Porting the compiler to support a new source language means you have to implement a compiler front end for that source language, *but you can reuse the optimizer and back end.*

The same applies for adding a new target architecture for the compiler. You just have to implement the back end and you can reuse the front end and optimizer.

Additionally, you can use specific parts of the compiler for other purposes. For example, pieces of the compiler front end could be used for documentation generation and static analysis tools.

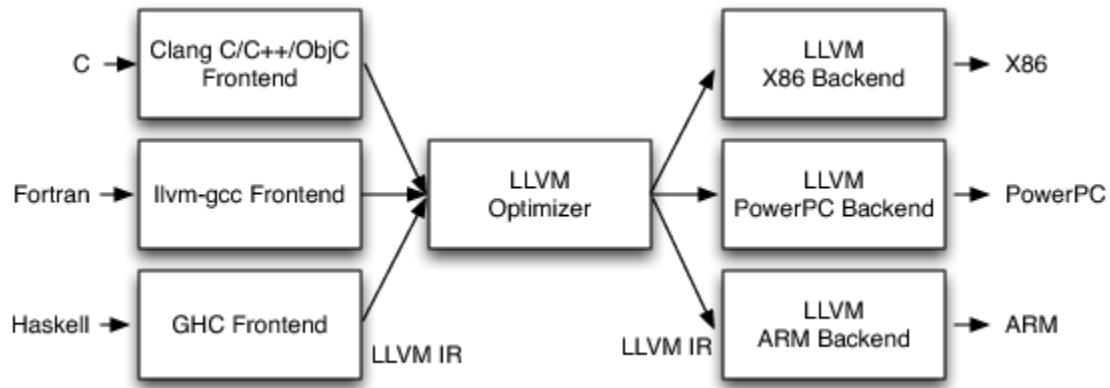
The main issue was that this model was rarely realized in practice. If you looked at the open source language implementations prior to LLVM, you'd see that implementations of Perl, Python, Ruby, Java, etc. shared no code.

While projects like GHC and FreeBASIC were designed to compile to multiple different CPUs, their implementations were specific to the one source language they supported (Haskell for GHC).

Compilers like GCC suffered from layering problems and leaky abstractions. The back end in GCC uses front end ASTs to generate debug info and the front end generates back end data structures.

The LLVM project sought to fix this.

LLVM's Implementation of the Three-Phase Design



In an LLVM-based compiler, the front end is responsible for parsing, validating, and diagnosing errors in the input code.

The front end then translates the parsed code into LLVM IR (LLVM Intermediate Representation).

The LLVM IR is a *complete code representation*. It is well specified and is the only interface to the optimizer.

This means that if you want to write a front end for LLVM, all you need to know is what LLVM IR is, how it works, and what invariants it expects.

LLVM IR is a low-level RISC-like virtual instruction set and it is how the code is represented in the optimizer. It generally looks like a weird form of assembly language.

Here's an example of LLVM IR and the corresponding C code.

```
define i32 @add1(i32 %a, i32 %b) {
entry:
%tmp1 = add i32 %a, %b
ret i32 %tmp1
}

define i32 @add2(i32 %a, i32 %b) {
entry:
%tmp1 = icmp eq i32 %a, 0
br i1 %tmp1, label %done, label %recurse

recurse:
%tmp2 = sub i32 %a, 1
%tmp3 = add i32 %b, 1
%tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
ret i32 %tmp4

done:
ret i32 %b
}
```

This LLVM IR corresponds to this C code, which provides two different ways to add integers:

```
unsigned add1(unsigned a, unsigned b) {
    return a+b;
}

// Perhaps not the most efficient way to add two numbers.
unsigned add2(unsigned a, unsigned b) {
    if (a == 0) return b;
    return add2(a-1, b+1);
}
```

Now, the optimizer stage of the compiler takes the LLVM IR and optimizes that code.

Most optimizations follow a simple three-part structure:

1. Look for a pattern to be transformed
2. Verify that the transformation is safe/correct for the matched instance
3. Do the transformation, updating the LLVM IR code

An example of an optimization is pattern matching on basic math expressions like replacing $X - X$ with 0 or $(X^2) - X$ with X .

You can easily customize the optimizer to add your own optimizing transformations.

After the LLVM IR code is optimized, it goes to the back end of the compiler (also known as the code generator).

The LLVM code generator transforms the LLVM IR into target specific machine code.

The code generator's job is to produce the best possible machine code for any given target.

LLVM's code generator splits the code generation problem into individual passes—instruction selection, register allocation, scheduling, code layout optimization, assembly emission, and more.

You can customize the back end and choose among the default passes (or override them) and add your own target-specific passes.

This allows target authors to choose what makes sense for their architecture and also permits a large amount of code reuse across different target back ends (code from a pass for one target back end can be reused by another target back end).

The code generator will output target specific machine code that you can now run on your computer.

For a more detailed overview and details on unit testing, modular design and future directions of LLVM, read the full [post](#)!

How WhatsApp served 1 billion users with only 50 engineers.

In 2016, WhatsApp reached more than a billion users and had the following load stats

- 42 billion messages sent daily
- 1.6 billion pictures sent daily
- 250 million videos sent daily

They managed to serve this scale with only 50 engineers.

Here's a dive into the engineering culture and tech stack that made this possible.

Engineering Culture

WhatsApp's Engineering culture consists of 3 main principles

1. Keep Things Small
2. Keep Things Simple
3. Have a Single Minded Focus on the Mission

Keep Things Small

WhatsApp consciously keeps the engineering staff small to only about 50 engineers.

Individual engineering teams are also small, consisting of 1 - 3 engineers and teams are each given a great deal of autonomy.

In terms of servers, WhatsApp prefers to use a smaller number of servers and vertically scale each server to the highest extent possible.

Their goal was previously to have 1 million users for every server (but that's become more difficult as they've added more features to the app and as users are generating more activity on a per-user basis).

Having a fewer number of servers means fewer things breaking down, which makes it easier for the team to handle.

The same goes for the software side where they limit the total number of systems and components in production.

That means fewer systems that have to be developed, deployed and supported.

There aren't many systems/components that are developed and then put into maintenance mode (to eventually become orphans until something goes wrong).

Keep Things Simple

WhatsApp uses the mantra *Just Enough Engineering*.

They avoid over-investing in systems and components.

Instead, they focus on building just enough for scalability, security and reliability.

One of the key factors when they make technical choices is “what is the simplest approach?”

Also, they avoid investing in automation unless it’s completely necessary.

Have a Single Minded Focus on the Mission

Product Design at WhatsApp is incredibly focused.

It’s dedicated to delivering a core communications app with a great UI.

They avoid extra bells and whistles and don’t implement features that aren’t *exclusively focused on core communications*.

The simpler product makes it much easier to maintain and scale.

Tech Stack

The tech stack revolves around 3 core components: Erlang, FreeBSD and SoftLayer.

Erlang

Erlang is the programming language of choice for WhatsApp's backend systems.

Erlang was designed for concurrency from the start, and fault tolerance is a first class feature of the language.

You can read more about Erlang's fault tolerance [here](#).

Developer productivity with Erlang is also extremely high. However, it is a functional language, so it takes a little getting used to if you're not familiar with the paradigm.

The language is *very* concise and it's easy to do things with very few lines of code.

The [OTP](#) (Open Telecom Platform) is a collection of open source middleware, libraries and tools for Erlang.

WhatsApp tries to avoid dependencies as much as possible, but they do make use of Mnesia, a distributed database that's part of OTP.

Erlang also brings the ability to hotswap code. You can take new application code and load it into a running application without restarting the application.

This makes the iteration cycle very quick and allows WhatsApp to release quick fixes and have extremely long uptimes for their services.

To see exactly how WhatsApp's backend is built with Erlang, you can watch this [talk](#) from 2018.

FreeBSD

FreeBSD is the OS WhatsApp uses for their servers.

The decision to use FreeBSD was made by the founders of WhatsApp, based on their previous experience at Yahoo!

The founders (and a lot of the early team) all used to be part of Yahoo!, where FreeBSD was used extensively.

To see exactly how WhatsApp uses FreeBSD, you can watch this [talk](#).

Just note, the talk is from 2014, so some things may be out of date now.

SoftLayer

SoftLayer is the hosting platform that WhatsApp was using in 2016.

They picked SoftLayer for two main reasons

1. The availability of FreeBSD as a first class operating system.
2. The ability to order and operate bare metal servers.

However, SoftLayer is owned by IBM (part of IBM public cloud), and WhatsApp has since moved off SoftLayer to use Facebook's infrastructure.

They made the transition in [2017](#).

You can view the full talk on WhatsApp's engineering [here](#).

Get more specific details from this [High Scalability post](#) on WhatsApp Engineering.

The Architecture of Uber's API gateway

This is an [article](#) on the technical components of Uber's API gateway.

Summary

When Uber's ride sharing app makes a request to the backend, the first point of contact is Uber's API gateway.

The API gateway provides a single point of entry for all of Uber's apps and gives a clean interface to access data, logic or functionality from back-end microservices.

The API gateway is the place to implement things like rate limiting, security auditing, user access blocking, protocol conversion, and more.

How does the API Gateway work

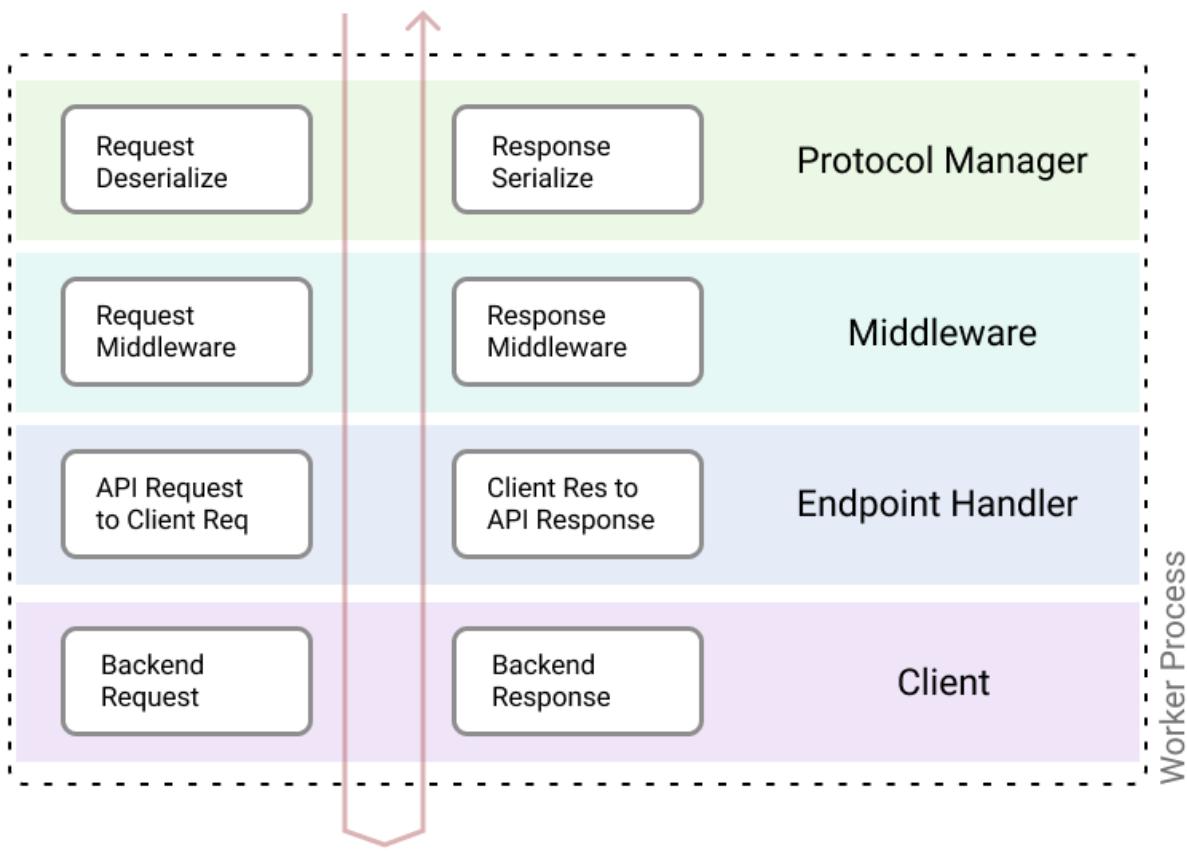
A backend engineer at Uber will be working on their own microservice (you can read about how Uber handles microservices [here](#)).

Their microservice will have an API with its own configuration parameters: path, type of request data, type of response, maximum calls allowed, apps allowed, observability, etc.

The engineer can then configure these parameters in a UI for Uber's API gateway. The UI walks the user through a step-by-step process for creating their API endpoint.

The gateway infrastructure will then convert these configurations into valid and functional APIs that can serve traffic from Uber's apps.

How a request flows through the API gateway



The four components are the Protocol Manager, Middleware, Endpoint Handler and finally the Client.

Each of the components operates on the request object on the way in and the same components are run in the reverse order on the response object's way out.

1. Protocol Manager - This is the first layer of the stack. It contains a deserializer and serializer for all of the protocols supported by the gateway. It can ingest any type of relevant protocol payload, including JSON, Thrift, or Protobuf.
1. Middleware - This layer handles things like rate limiting, authentication and authorization, etc. Each endpoint can choose to configure one or more middleware. If a middleware fails execution, the call short circuits the remainder of the stack and the response from the middleware will be returned to the caller.

Middleware is configured in a YAML file.

```
middlewares:  
- name: authentication  
  options:  
    path: header.x-user-uuid  
- name: transformRequest  
  options:  
    transforms:  
    - from: region  
      to: request.regionID
```

1. Endpoint Handler - This layer is responsible for request validation, payload transformation and converting the endpoint request object to the client request object based on the configured schema and serialization.
2. Client - This layer performs the request to the specific backend microservice. Clients are protocol-aware and generated based on the protocol selected during configuration.

The [full blog post](#) delves more deeply into each of these layers (and how users configure settings in the API gateway) and also talks about challenges faced and lessons learned.

If you'd like to read about how Uber thinks about scaling this API gateway, here's another interesting [blog post on that](#).

Map Reduce Explained

History behind Map Reduce

The MapReduce [paper](#) was first published in 2004 by Jeff Dean and Sanjay Ghemawat. It was originally designed, built and used by Google.

At the time, Google had an issue. Google's search engine required constant crawling of the internet, content indexing of every website and analyzing the link structure of the web (for the PageRank algorithm).

This means massive computations on terabytes of data.

Running this scale of computations on a single machine is obviously impossible, so Google bought thousands of machines that could be used by engineers.

For a while, engineers had to laboriously hand-write software to take whatever problem they're working on and pharm it out to all the computers.

Engineers would have to manage things like parallelization, fault tolerance, data distribution and various other concepts in distributed systems.

If you're not skilled in distributing systems, then it can be really difficult to do this. If you are skilled in distributed systems, then you can do it but it's a waste of your time to do it again and again.

Therefore, Google wanted a framework that allowed engineers to focus on the code for their problem (building a web index/link analyzer/whatever) and provided an interface so engineers could use the vast array of machines without having to worry about the distributed systems stuff.

The framework they came up with is MapReduce.

Map and Reduce functions

MapReduce is based on two functions from the Lisp programming language (and many other functional languages): Map and Reduce (also known as Fold).

Map takes in a list of elements (or any iterable object) and a function, applies the function to each element in the list and then returns the list.

Here's an example of using *map* to square the numbers in a list.

```
● ● ●  
nums = [1,2,3,4,5]  
squared = list(map(lambda x: x*x, nums))  
print(squared) # [1, 4, 9, 16, 25]
```

Reduce takes in a list of elements (or any iterable object), a function and a starting element.

The function (that gets passed into the *Reduce* operation) takes in a starting element and an element from the list and combines the two in some way. It then returns the combination.

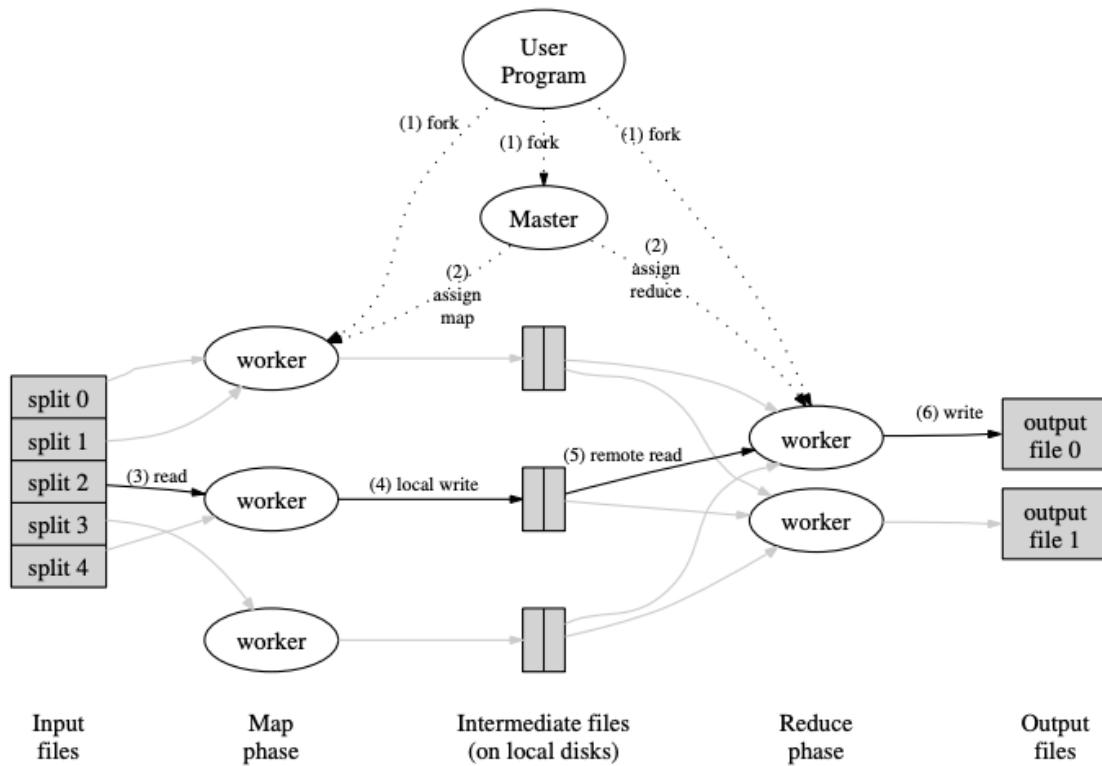
The *reduce* operation sequentially runs the function on all the elements in the list combining each element with the result from the previous function call.

Here's an example of using *reduce* to find the sum of elements in a list.



```
nums = [1,2,3,4,5]
print(reduce(lambda x,y: x + y, nums)) # 15
```

How MapReduce Works



Example

You have a billion documents and you want to create a dictionary of all the words that appear in those documents and the count of each word (number of times each word appears across the documents).

In order to do this with MapReduce, you have to write a map function and a reduce function.

The map function could look something like this.

```
map(String key, String value):  
    // key: document name  
    // value: document contents  
    for each word w in value:  
        EmitIntermediate(w, "1")
```

EmitIntermediate is a function provided by the MapReduce framework. It's where you send the output of your map function.

The map function emits each word plus a count of occurrences (here it's just 1).

The reduce function will then sum together all counts emitted for a particular word.

Here's the reduce function.

```
reduce(String key, Iterator values):  
    // key: a word  
    // values: a list of counts  
    int result = 0;  
    for each v in values:  
        result += ParseInt(v);  
    Emit(AsString(result));
```

In order to use the MapReduce framework (and take advantage of the distributed computers), you'll have to provide the map function (listed above), the reduce function (also listed above), names of the input files (for the billion documents), output files (where you want MapReduce to put the finished dictionary) and some tuning parameters (discussed below).

If you'd like to see the full program for this using the MapReduce framework, please look at page 13 of the [MapReduce paper](#).

Now, here's a breakdown of exactly how the MapReduce framework works internally.

The MapReduce library will look at the input files and split them into M pieces (M is a parameter specified by the user).

It will then start up many copies of the MapReduce program on a cluster of machines.

One of the copies will be the master. The rest of the copies are workers that are assigned tasks by the master.

There are M map tasks (for each piece of the input files) and R reduce tasks (R is specified by the user) that the master will assign to the workers. The master will pick idle workers and assign them map or reduce tasks.

A worker who is assigned a map task will read the contents of their input split. They will parse key/value pairs out of the input data and pass *each* key/value pair to the user-provided Map function.

The Map function will output *intermediate* key/value pairs.

The intermediate key/value pairs are partitioned into R regions by the *partitioning function*. The location of these pairs is passed back to the master.

The default partitioning function just uses hashing (e.g. $\text{hash}(\text{key}) \bmod R$) but the user can provide a special partitioning function if desired. For example, the output keys could be URLs and the user wants URLs from the same website to go to the same output file. Then, the user can specify his own partitioning function like $\text{hash}(\text{Hostname(urlkey)}) \bmod R$.

After the location of the intermediate key/value pairs is passed to master, the master assigns reduce tasks to workers and notifies them of the storage locations for their assigned key/value pairs (one of the R partitions).

The reduce worker will read the intermediate key/value pairs from their region and then sort them by the intermediate key so that all occurrences of the same key are grouped together.

The reduce worker then iterates over the sorted intermediate data.

For each unique intermediate key encountered, the reduce worker passes the key and the corresponding set of intermediate values to the user's Reduce function.

The output of the Reduce function is appended to a final output file for this reduce partition.

After all the map tasks and reduce tasks have been completed, the master wakes up the user program.

The output of the MapReduce execution will be available in R output files, one for each of the Reduce workers.

You can read the full MapReduce paper [here](#).

Building a basic Storage Engine

We're going to dive into the most important component of a database, the storage engine.

The storage engine is responsible for storing, retrieving and managing data in memory and on disk.

A DBMS will often let you pick which storage engine you want to use. For example, MySQL has several choices for the storage engine, including RocksDB and InnoDB.

Having an understanding of how storage engines work is crucial for helping you pick the right database.

The two most popular types of database storage engines are

- Log Structured (like Log Structured Merge Trees)
- Page Oriented (like B-Trees)

Log Structured storage engines treat the database as an append-only log file where data is sequentially added to the end of the log.

Page Oriented storage engines break the database down into *fixed-size pages* (traditionally 4 KB in size) and they read/write one page at a time.

Each page is identified using an address, which allows one page to refer to another page. These page references are used to construct a tree of pages.

In this update, we'll be focusing on Log Structured storage engines. We'll look at how to build a basic log structured engine with a hash index for faster reads.

Log Structured Storage Engines

Our storage engine will handle key-value pairs. It'll have two functions

- `db_set(key, value)` - give the database a (key, value) pair and the database will store it. If the key already exists then the database will just update the value.
- `db_get(key)` - give the database a key and the database will return the associated value. If the key doesn't exist, then the database will return null.

The storage engine works by maintaining a log of all the (key, value) pairs.

By a log, we mean an append-only sequence of records. The log doesn't have to be human readable (it can be binary).

Anytime you call `db_set`, you append the (key, value) pair to the bottom of the log.

This is done *regardless* of whether the key already existed in the log.

The append-only strategy works well because appending is a sequential write operation, which is generally much faster than random writes on magnetic spinning-disk hard drives (and on solid state drives to some extent).

Now, anytime you call `db_get`, you look through all the (key, value) pairs in the log and return the *last* value for any given key.

We return the last value since there may be duplicate values for a specific key (if the key, value pair was inserted multiple times) and the last value that was inserted is the most up-to-date value for the key.

The `db_set` function runs in $O(1)$ time since appending to a file is very efficient.

On the other hand, `db_get` runs in $O(n)$ time, since it has to look through the entire log.

This becomes an issue as we scale the log to handle more (key, value) pairs.

But, we can make `db_get` faster by adding a database index.

The general idea behind an index is to *keep additional metadata on the side*.

This metadata can act as a “signpost” as the storage engine searches for data and helps it find the data faster.

The database index is an *additional* structure that is derived from the primary data and adding a database index to our storage engine will not affect the log in any way.

The tradeoff with a database index is that your database reads can get significantly faster. You can use the database index during a read to achieve sublinear read speeds.

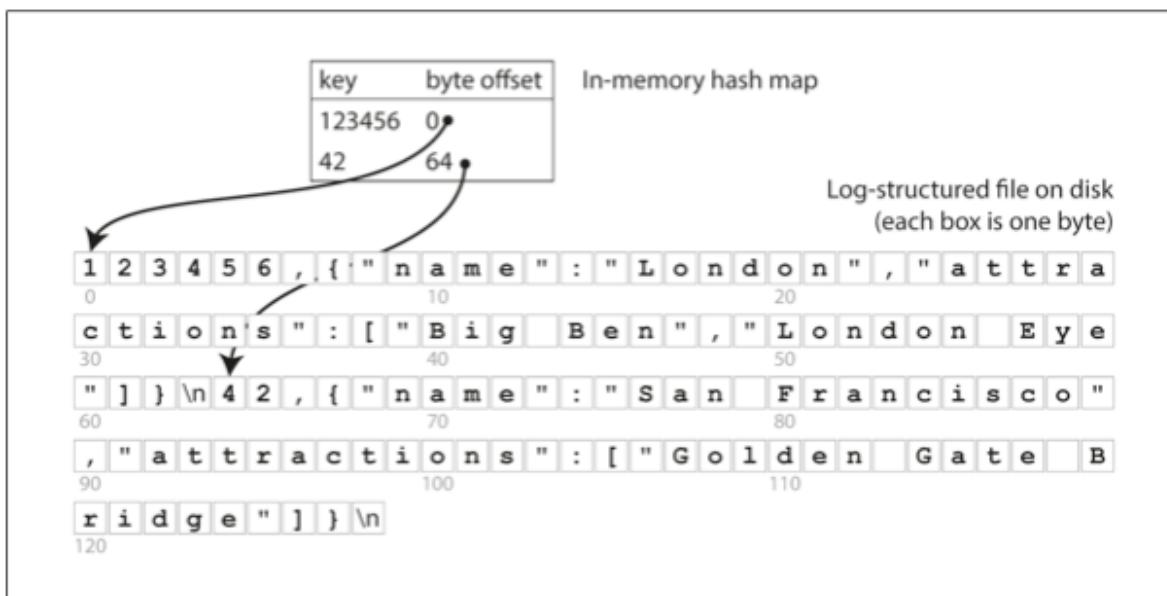
However, database writes are slower now because for every write you have to update your log and update the database index.

An example of a database index we can add to our storage engine is a hash index.

Hash Indexes

One index we can use is a hash table.

We keep a hash table in-memory where each key in our log is stored as a key in our hash table and the location in the log where that key can be found (the byte offset) is stored as the key's value in our hash table.



Credits to *Designing Data Intensive Applications* for the Image

Now, the process for db_set is

1. Append the (key, value) pair to the end of the log. Note the byte offset for where it's added.
2. Check if the key is in the hash table.
3. If it is, then update its value in the hash table to the new byte offset.
4. If it isn't, then add (key, byte offset) to the hash table.

One requirement with this strategy is that all your (key, byte offset) pairs have to fit in RAM since the hash map is kept in-memory.

An issue that will eventually come up is that we'll run out of disk space.

Since we're only appending to our log, we'll have repeat entries for the same key even though we're only using the most recent value.

We can solve this issue by breaking our log up into segments.

Whenever our log reaches a certain size, we'll close it and then make subsequent writes to a new segment file.

Then, we can perform compaction on our old segment, where we throw away duplicate keys in that segment and only keep the most recent update for each key.

After removing duplicate keys, we can merge past segments together into larger segments of a fixed size.

We'll also create new hash tables with (key, byte offset) pairs for all the past merged segments we have and keep those hash tables in-memory.

The merging and compaction of past segments is done in a background thread, so the database can still respond to read/writes while this is going on.

Now, the process for db_get is

1. Check the current segment's in-memory hash table for the key.
2. If it's there, then use the byte offset to find the value in the segment on disk and return the value.
3. Otherwise, look through the hash tables for our previous segments to find the key and its byte offset.

The methodology we've described is very similar to a real world storage engine - Bitcask.

Bitcask is a storage engine used for the Riak database, and it's written in Erlang. You can read Bitcask's whitepaper [here](#).

Bitcask (the storage engine that uses this methodology) also stores snapshots of the in-memory hash tables to disk.

This helps out in case of crash recovery, so the database can quickly spin back up if the in-memory hash tables are lost.

This is a brief overview of how a Log Structured database can work.

However, issues that arise are

- The hash table of (key, byte offsets) must fit in memory. Maintaining a hashmap on disk is too slow.
- Range queries are not efficient. If you want to scan over all the keys that start with the letter *a*, then you'll have to look up each key individually in the hash maps.

Robinhood's Tech Stack

Robinhood is a Stock Brokerage app founded in 2013. It charges zero commissions on trades of stocks and ETFs and is also mobile-first. The majority of users place trades from their mobile devices.

Robinhood's Tech Stack

Robinhood is cloud-native, based on AWS.

Robinhood paid \$60 million dollars to AWS in 2020 for cloud hosting fees. This is a *5x increase* from their 2019 cloud bill of \$12 million dollars.

If you'd like to learn more about Robinhood's engineering, check out their interview on the [Software Engineering Daily](#) podcast. We'll summarize the interesting bits here.

When Robinhood was first getting started, they were a Python/Django shop, however they've been shifting towards Go.

They've also been microservices oriented, and most of their APIs are written in Python and Go. There is some Java and Rust in their codebase however.

Robinhood is built on AWS, so they use Amazon RDS (Relational Database Service) for their data store. They use [Postgres](#) as the database engine for RDS.

PaaS vs. DIY

Since they're on AWS, an interesting choice that comes up is whether they should utilize AWS's PaaS (Platform as a Service) offerings or if they should go the "Do It Yourself" route.

An example is Elasticsearch. At first, Robinhood utilized Amazon Elasticsearch Service and went the PaaS route. However, their specific workload wasn't well suited for AWS's PaaS offering.

Therefore, they made the decision to deploy Elasticsearch on an EC2 instance and modify the database so it suited their workload (Elasticsearch is [open source](#)).

Jaren Glover (tech lead at Robinhood) found that PaaS products work great when you play inside the guardrails in which they're presented.

However, if you move from a generic compute to something domain specific, then you may start to bump against those guardrails and not have the quality of service that you want.

Another interesting challenge that comes up when building a Stock brokerage is how you manage time.

It's *very* important to process orders *in the same order that the customer placed them* and also to route orders correctly relative to which customer placed which order first.

In order to do this, Robinhood relies on NTP - [Network Time Protocol](#). NTP allows you to synchronize clocks between computers over a variable-latency network.

NTP works in a client-server type model, but can also be adapted for a peer-to-peer system.

NTP can usually maintain time to within tens of milliseconds over the public Internet, and can achieve better than one millisecond accuracy over LANs.

Robinhood (and other brokerages) are required to ensure their trading computers are synced with atomic clocks maintained by the National Institute of Standards and Technology (NIST), a non-regulating agency of the US Department of Commerce.

How Robinhood Scaled

Robinhood, despite being a brokerage, has experienced viral growth. The app is frequently ranked #1 in the iOS and Android app stores, a spot that is usually reserved for some type of social media app (Instagram, Snapchat, YouTube, etc.). It's unheard of for a *financial app* to get that kind of growth.

In December 2019, Robinhood's system received 100k requests per second at peak time. In June 2020, Robinhood's system received 750k requests per second, a *7x load increase to the system in a 6 month period.*

To respond to this, Robinhood set out to make their brokerage infrastructure horizontally scalable. They accomplished this by implementing *sharding* into their system *at the application layer* (not just the database).

Sharding is where you split up a large dataset into smaller partitions (shards).

Robinhood created multiple shards where each shard held a subset of their users and every shard had its own application servers, database and deployment pipeline.

In order to make the divided system appear as one system (rather than independent shards), Robinhood built out several new layers.

- *Routing Layer* - the routing layer handles routing external API requests to the correct shards. The layer first inspects and maps the request to a specific user. Then, it makes a synchronous API call to Robinhood's shard mapping service to look up the shard ID for the user. After, it sends the request to the correct application server with that shard.
- *Aggregation Layer* - the aggregation layer is an intermediary layer for internal API traffic. It enables other services to query data without knowing which shard it lives in, by joining data from multiple shards. It is a stateless service that fans our requests across shards and then merges the results and returns them.
- *Kafka Message Streaming* - Robinhood uses Kafka streams for backend services to send messages to each other. However, Robinhood has to ensure that messages are consumed by the correct shard. To ensure this, Robinhood has every shard consume *all messages*, but only process messages that correspond to users that exist in that shard.

[You can read more about Robinhood's horizontal scaling efforts here.](#)

How Slack Designs APIs

It's extremely important to carefully think about your API design from the beginning.

Designing a bad API means wasting developer time and (for a public API) poor adoption.

It's also extremely difficult to make breaking changes to an API after shipping so bad API design can be hard to fix.

In order to avoid this at Slack, they've [published some principles around API design](#).

Slack's API Design Principles

1. Do one thing and do it well - It can be tempting to try and solve too many problems at once. Instead, pick a specific use case and design your API around solving that. Simple APIs are more easy to understand and easier to scale. *It's easy to add features to an API, but hard to remove them.*
2. Make it fast and easy to get started - Developers should be able to complete a basic task using your API quickly. At Slack, they want entry-level developers to be able to learn about the platform, create an app, and send their first API call within 15 minutes.
3. Strive for intuitive Consistency - Developers should be able to guess parts of your API without reading the documentation. You can make your API more intuitive with your choices for endpoint names, input parameters and output responses. Adhere closely with industry standards and also make sure your API is consistent with your product. You should choose field names based on what you call those concepts in your product.
4. Return meaningful errors - Good error messages are easy to understand, unambiguous and actionable. Implementation details should not leak in your error messages.

5. Design for scale and performance - While designing an API, you should follow best practices to avoid bad performance. Implement things like
 1. Pagination
 2. Rate Limiting
6. Avoid breaking changes - A breaking change is any change that can stop an existing client app from functioning as it was before the change. Avoid these and have an apologetic communication plan if you need to make a breaking change.

How Notion sharded their Postgres Database

[Notion](#) is an app that is meant to serve as a personal (or corporate) workspace.

You can store notes, tasks, wikis, kanban boards and other things in a Notion workspace and you can easily share it with other users.

If you've been a Notion user for a while, you probably noticed that the app got [extremely slow in late 2019 and 2020](#).

Earlier this year, Notion sharded their Postgres monolith into a fleet of horizontally scalable databases. The resulting performance boost was pretty big.

Sharding a database means partitioning your data across multiple database instances.

This allows you to run your database on multiple computers and scale horizontally instead of vertically.

When to Shard?

Sharding your database prematurely can be a big mistake. It can result in an increased maintenance burden, new constraints in application code and little to no performance improvement (so a waste of engineering time).

However, Notion was growing extremely quickly, so they knew they'd have to implement sharding at some point.

The breaking point came when the Postgres [VACUUM](#) process began to stall consistently.

The VACUUM process clears storage occupied by dead tuples in your database.

When you update data in Postgres, the existing data is *not* modified. Instead, a new (updated) version of that data is added to the database.

This is because it's not safe to directly modify existing data, as other transactions could be reading it.

This is called Multiversion Concurrency Control (MVCC).

At a later point, you can run the VACUUM process to delete the old, outdated data and reclaim disk space.

If you don't regularly vacuum your database (or have Postgres run autovacuum, where it does this for you), you'll eventually reach a transaction ID wraparound failure.

So, you *must* vacuum your database or it will eventually fail.

Having the VACUUM process consistently stall is not an issue that can be ignored.

Application-Level vs. Managed

Sharding can be divided into two approaches

- Application-Level Sharding - You implement the data partitioning scheme *in your application code*. You might direct all American users to one database and all Asian users to another database.
- Third-Party Sharding - You rely on a third party to handle the sharding for you. An example is Citus, an open source extension for Postgres.

Notion decided to go with Application-Level sharding.

They didn't want to go with a third party solution because they felt it's sharding logic would be opaque and hard to debug.

Shard Key

In order to shard a database, you have to pick a shard key. This determines how your data will be split up amongst the shards.

You want to pick a shard key that will equally distribute loads amongst all the shards.

If one shard is getting a lot more reads/writes than the others, that can make scaling very difficult.

Notion decided to partition their database by workspace. Workspaces are the folders that contain all the pages, tasks, notes, etc.

So, if you're a student using Notion, you might have separate Workspaces for all your classes.

Each workspace is assigned a UUID upon creation, so that UUID space is partitioned into uniform buckets.

Each bucket goes to a different shard.

How many Shards?

Notion ended up going with 460 logical shards distributed across 32 physical databases (with 15 logical shards per database).

This allows them to handle their existing data and scale for the next two years (based off their projected growth).

Database Migration

After establishing how the sharded database works, you still have to migrate from the old database to the new distributed database.

1. Double-write: Incoming writes are applied to both the old and new databases.
2. Backfill: Migrate the old data to the new database.
3. Verification: Ensure the integrity of data in the new database.
4. Switch-over: Actually switch to the new database. This can be done incrementally, e.g. double-reads, then migrate all reads.

[Read the full details in the blog post!](#)

Google File System

In 1998, the first Google index had [26 million pages](#). In 2000, the Google index reached a billion web pages. By 2008, Google was processing more than 1 trillion web pages.

As you might imagine, the storage needs required for this kind of processing were massive and rapidly growing.

To solve this, Google built Google File System (GFS), a scalable distributed file system written in C++. Even in 2003, the largest GFS cluster provided hundreds of terabytes of storage across thousands of machines and it was serving hundreds of clients concurrently.

GFS is a proprietary distributed file system, so you'll only encounter it if you work at Google. However, Doug Cutting and Mike Cafarella implemented Hadoop Distributed File System (HDFS) based on Google File System and HDFS is used widely across the industry.

LinkedIn recently published a [blog post on how they store 1 exabyte of data across their HDFS clusters](#). An exabyte is 1 billion gigabytes.

In this post, we'll be talking about the goals of GFS and its design. If you'd like more detail, you can read the full GFS paper [here](#).

Goals of GFS

The main goal for GFS was that it be *big* and *fast*. Google wanted to store extremely large amounts of data and also wanted clients to be able to quickly access that data.

In order to accomplish this, Google wanted to use a distributed system built of inexpensive, *commodity machines*.

Using commodity machines is great because then you can quickly add more machines to your distributed system (as your storage needs grow). If Google relied on specialized hardware, then there may be limits on how quickly they can acquire new machines.

To achieve the scale Google wanted, GFS would have to use thousands of machines. When you're using that many servers, you're going to have *constant* failures. Disk failures, network partitions, server crashes, etc. are an everyday occurrence.

Therefore, GFS needed to have systems in place for *automatic failure recovery*. An engineer shouldn't have to get involved every time there's a failure. The system should be able to handle common failures on its own.

The individual files that Google wanted to store in GFS are quite big. *Individual files are typically multiple gigabytes* and so this affected the block sizes and I/O operation assumptions that Google made for GFS.

GFS is designed for *big, sequential reads and writes* of data. Most files are mutated by *appending new data* rather than overwriting existing data and *random writes within a file are rare*. Because of that access pattern, appending new data was the focus of performance optimization.

Design of GFS

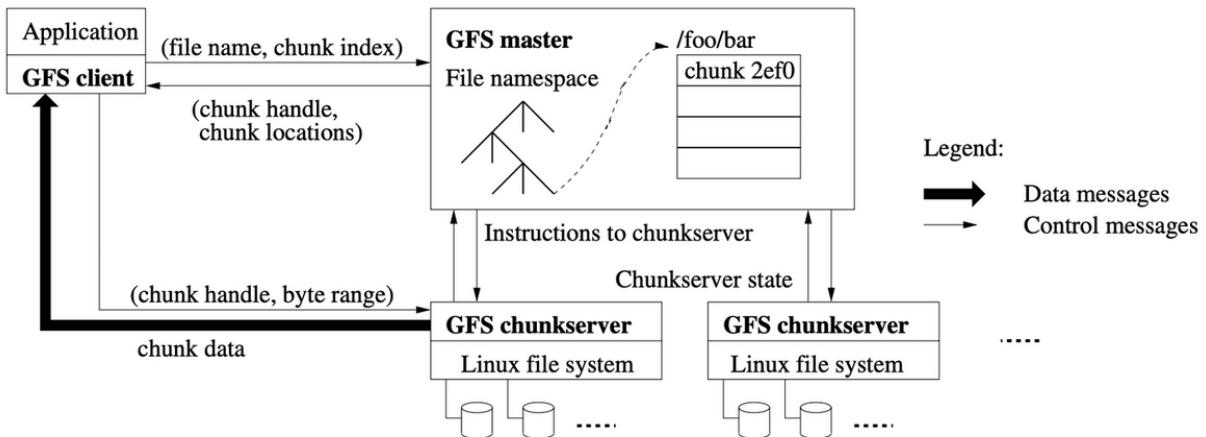


Figure 1: GFS Architecture

A GFS cluster consists of a single *master node* and multiple *chunkserver nodes*.

The master node maintains the file system's metadata and coordinates the system. The chunkserver nodes are where all the data is stored and accessed by clients.

Files are divided into 64 megabyte chunks and assigned a 64 bit chunk handle by the master node for identification. The chunks are then stored on the chunkservers with each chunk being replicated across several chunkservers for reliability and speed (the default is 3 replicas).

The master node keeps track of the file namespace, the mappings from files to chunks and the locations of all the chunks. It also handles garbage collection of orphaned chunks and chunk migration between the chunkservers. The master periodically communicates with all the chunkservers through HeartBeat messages to collect its state and give it instructions.

An interesting design choice is the decision to use a *single* master node. Having a single master greatly simplified the design since the master could make chunk placement and replication decisions without coordinating with other master nodes.

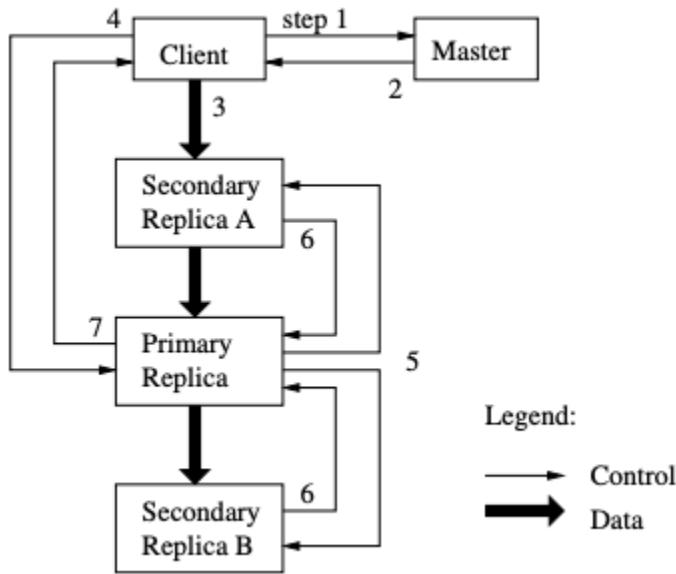
However, Google engineers had to make sure that the single master node doesn't become a bottleneck in the system.

Therefore, clients never read or write file data through the master node. Instead, the client asks the master which chunkservers it should contact. Then, the client caches this information for a limited time so it doesn't have to keep contacting the master node.

GFS Mutations

A mutation is an operation that changes the contents or the metadata of a chunk (so a write or an append operation).

In order to guarantee consistency amongst the replicas after a mutation, GFS performs mutations in a certain order.



As stated before, each chunk will have multiple replicas. The master will designate *one* of these replicas as the *primary replica*.

Here are the steps for performing a mutation to a chunk:

1. The client asks the master which chunkserver is the primary chunk and for the locations of the other chunkservers that have that chunk.
2. The master replies with the identity of the primary chunkserver and the other replicas. The client caches this information.
3. The client pushes data directly to all the chunkserver replicas. Each chunkserver will store the data in an internal LRU buffer cache.

4. Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary chunkserver. The primary chunkserver then applies the mutations to its state.
5. The primary chunkserver forwards the write requests to the other chunkservers. Each chunkserver then applies the mutation to their state.
6. The secondary chunkservers all reply to the primary chunkserver indicating that they've completed the operation.
7. The primary chunkserver replies to the client informing the client that the write was successful (or if there were errors).

GFS Interface

GFS organizes files hierarchically in directories and identifies them by pathnames, like a standard file system. The master node keeps track of the mappings between files and chunks.

GFS provides the usual operations to create, delete, open, close, read and write files.

It also has snapshot and record append operations.

Snapshot lets you create a copy of a file or directory tree at low cost.

Record append allows multiple clients to append data to a file concurrently and it guarantees the atomicity of each individual client's append.

To learn more about Google File System, read the full paper [here](#).

If you'd like to read about the differences between GFS and HDFS, you can check that out [here](#).

Scaling an API with Rate Limiters

Stripe Engineering wrote a fantastic [blog post](#) on how they think about rate limiters.

Here's a summary.

[Rate Limiting](#) is a technique used to limit the amount of requests a client can send to your server.

It's incredibly important to prevent DoS attacks from clients that are (accidentally or maliciously) flooding your server with requests.

A rule of thumb for when you should use a rate limiter is *if your users can reduce the frequency of their API requests without affecting the outcome of their requests, then a rate limiter is appropriate.*

For example, if you're running Facebook's API and you have a user sending 60 requests a minute to query for their list of Facebook friends, you can rate limit them without affecting their outcome. It's unlikely that they're adding new Facebook friends every single second.

Rate Limiting is great for day-to-day operations, but you'll occasionally have incidents where some component of your system is down and you can't process requests at your normal level.

In these scenarios, [Load Shedding](#) is a technique where you drop low-priority requests to make sure that critical requests get through.

Stripe is a payment processing company (you can use their API to collect payments from your users) so a critical request for them is a request to create a charge.

An example of a non-critical method would be a request to read charge data from the past.

Stripe uses 4 different types of limiters in production (2 rate limiters and 2 load shedders).

Request Rate Limiter

Restricts each user to n requests per second. However, they also built in the ability for a user to briefly burst above the cap to handle legitimate spikes in usage.

Concurrent Requests Limiter

Restricts each user to n API requests in progress *at the same time*.

This helps stripe manage the load of their CPU-intensive API endpoints.

Fleet Usage Load Shedder

Stripe divides their traffic into two types: critical API methods and non-critical methods.

An example of a critical method would be creating a charge (charging a customer for something), while a non-critical method is listing a charge (looking at past charges).

Stripe always reserves a fraction of their infrastructure for critical requests. If the reservation number is 10%, then any non-critical request over the 90% allocation would be rejected with a 503 status code.

Worker Utilization Load Shedder

Stripe uses a set of workers to independently respond to incoming requests in parallel. If workers start getting backed up with requests, then this load shedder will shed lower priority traffic.

Stripe divides their traffic into 4 categories

- Critical Methods
- POSTs
- GETs
- Test mode traffic (traffic from developers testing the API and making sure payments are properly processed)

If worker capacity goes below a certain threshold, Stripe will begin shedding less-critical requests, starting from test mode traffic.

Building a rate limiter in practice

There are quite a few algorithms you can use to build a rate limiter. Algorithms include

Token Bucket - Every user gets a bucket with a certain amount of “tokens”. On each request, tokens are removed from the bucket. If the bucket is empty, then the request is rejected.

New tokens are added to the bucket at a certain threshold (every n seconds). The bucket can hold a certain number of tokens, so if the bucket is full of tokens then no new tokens will be added.

Fixed Window - The rate limiter uses a window size of n seconds for a user. Each incoming request from the user will increment the counter for the window. If the counter exceeds a certain threshold, then requests will be discarded.

After the n second window passes, a new window is created.

Sliding Log - The rate limiter tracks every user’s request in a time-stamped log. When a new request comes in, the system calculates the sum of logs to determine the request rate. If the request rate exceeds a certain threshold, then it is denied.

After a certain period of time, previous requests are discarded from the log.

Stripe uses the token bucket algorithm to do their rate limiting.

GitHub's transition from Monolith to Microservices

Sha Ma is the VP of Software Engineering at GitHub.

She gave a talk at Qcon 2020 about GitHub's transition from a Monolith architecture to Microservices-oriented architecture.

You can view the full talk and transcript [here](#).

Here's a summary

History

GitHub was founded in 2008 by Chris Wanstrath, P.J. Hyett, Tom Preston-Werner and Scott Chacon.

The founders of the company were open source contributors and influencers in the Ruby community. Because of this, GitHub's architecture is deeply rooted in Ruby on Rails.

With the Ruby on Rails monolith, GitHub scaled to 50 million developers on the platform, over 100 million repositories and over 1 billion API calls per day.

Over the past 18 months, GitHub has grown rapidly as a company. They've doubled the number of engineers at the company, and now have over 2000 employees.

The company is also highly distributed with over 70% of employees working outside of the headquarters, working in all timezones.

Because of this diversity of engineers, GitHub is having trouble scaling the monolith.

Having everyone learn Ruby before they can be productive and having everyone doing development in the same monolithic code base is no longer the most efficient way to scale GitHub.

Therefore, GitHub engineering took a deep look at a Microservices architecture.

Here are some of the pros GitHub saw for a Monolith and Microservice architecture.

Pros of a Monolith architecture

- Infrastructure Simplicity - A new employee can get GitHub up and running on their local machine within hours.
- Code Simplicity - You don't have to add extra logic to deal with timeouts or worry about failing gracefully due to network latency and outages.
- Architecture & Organization simplicity - Everyone has familiarity with the same codebase and it's easier to move people around to work on different features within the monolith.

Pros of a Microservice architecture

- System ownership - There are functional boundaries for teams through clearly defined API contracts. This gives teams much more ownership over their features and also gives them freedom to choose the tech stack that makes the most sense for them. They just have to make sure the API contract is followed.
- Separation of Concerns - Quicker ramp-up time for a new developer joining a team since a developer no longer has to understand all the inner workings of a large monolithic code base in order to be productive.
- Scaling separately - Services can now be scaled separately based on their individual needs

Based on these tradeoffs, GitHub decided to shift to a Microservices-oriented architecture.

However, the change isn't expected to be immediate or rapid. For the foreseeable future, GitHub plans to have a hybrid monolith-microservices environment.

For this reason, it's important for them to continue to maintain and improve the monolith codebase.

How to break up the Monolith

The first step towards breaking up a monolith is to think about the separation of code and data based on feature functionalities.

This can be done within the monolith before physically separating them into a microservices environment. It's generally a good architectural practice to make the codebase more manageable.

Start with the data and pay close attention to how it's being accessed.

Each service should own and control access to its own data. Data access should only happen through clearly defined API contracts.

If you don't enforce this, you can fall into a common microservice anti-pattern: the distributed monolith.

This is where you have the inflexibility of a monolith *and* the complexity of microservices.

Separating Data

Before making the transition to microservices, GitHub made sure they got data separation right. Getting it wrong can lead to the distributed monolith anti-pattern.

They first looked at their monolith and identified the functional boundaries within the database schemas.

Then, they grouped the actual database tables along these functional boundaries.

They grouped together everything related to repositories, everything related to users, and everything related to projects. The resulting functional groups are referred to as *schema domains*.

The repository schema domain holds all repository information like pull requests, issues, review comments, etc.

Then, GitHub implemented a query watcher in the monolith to detect and alert them anytime a query crosses multiple schema domains.

If a query touched more than one schema domain, then they would break the query up and rewrite it into multiple queries that respect the functional boundaries. They would then perform the necessary joins at the application layer.

Separating Services

When separating services out of the monolith to a microservice, you should start with the core services and then work your way out to the feature level.

Dependency direction should always go from *inside* of the monolith to *outside* of the monolith, NOT the other way around. If you have dependency directions from microservices to inside the monolith then that can lead to the distributed monolith anti-pattern.

At GitHub, the core service that they extracted first was Authentication and Authorization. The Rails monolith communicated with the microservice using Twirp, a gRPC-like service-to-service communications framework, with an inside-to-outside dependency direction (inside of the monolith to outside of the monolith).

When separating services out of the monolith, you should be on the lookout for things that keep developers working in the monolith.

A common example is shared tooling that is built over time and makes development inside the monolith more convenient. Make those shared resources available to developers outside of the monolith.

An example at GitHub was feature flags that provide monolith developers an easy way to control who sees a new feature.

Finally, make sure to remove old code paths once the new services are up and running. Have a plan to move 100% of the traffic over to the new service, so you don't get stuck supporting two sets of code forever.

For more details, you can view the full talk [here](#).

Partitioning GitHub's Relational Database

In our last summary (included below), we talked about GitHub's transition from a monolith to microservices.

This summary is on GitHub's [blog post](#) on how they built tooling to make partitioning their database easier.

Until recently, GitHub was built around one MySQL database cluster that housed a large portion of the data used for repositories, issues, pull requests, user profiles, etc.

This MySQL cluster was called *mysql1*.

This created challenges around scaling and issues around reliability since all of GitHub's core features would stop working if the database cluster went down.

In 2019, GitHub set up a plan to improve their ability to partition their relational databases. The goal was to create better tooling that improved GitHub's ability to partition relational databases.

The tooling GitHub built made partitioning much easier by allowing for

- *Virtual Partitions* - Engineers can separate database tables virtually in the application layer so that the physical separation process is easier.
- *Move data without downtime* - After the database is virtually partitioned, Engineers can easily move partitioned database tables to a different database cluster without downtime.

Since implementing these changes, GitHub has seen a significant decrease in load on the main database cluster.

In 2019, *mysql1* answered 950,000 queries per second on average, 900,000 queries per second on replicas, and 50,000 queries per second on the primary.

In 2021, the same database tables were partitioned over several database clusters. The average load on each host *halved* despite the total queries per second *increasing* to 1,200,000 queries per second.

Here's a rundown on the technical changes GitHub made to implement Virtual Partitions and moving data without downtime.

Virtual Partitions

Before database tables are physically partitioned, they need to be virtually partitioned in the application layer. You can't have SQL queries that span partitioned (or soon to be partitioned) database tables.

Schema Domains

In order to implement Virtual Partitioning, GitHub first created schema domains, where a schema domain describes a tightly coupled set of database tables that are frequently used together in queries.

An example of a schema domain is the *gists* schema domain, which consists of the tables *gists*, *gist_comments*, and *starred_gists*. These tables would remain together after a partition.

GitHub stored a list of all the schema domains in a YAML configuration file. Here's an example of a YAML file with the gists, repositories and users schema domains and their respective tables.

```
gists:
  - gist_comments
  - gists
  - starred_gists
repositories:
  - issues
  - pull_requests
  - repositories
users:
  - avatars
  - gpg_keys
  - public_keys
  - users
```

Now, GitHub needed to enforce these schema domains. They want to make sure that application code doesn't have SQL queries or transactions that span schema domains.

They enforce this with SQL Linters.

SQL Linters

GitHub has two SQL linters (a Query linter and a Transaction linter) that enforce virtual boundaries between the schema domains.

They identify any violating queries and transactions that span schema domains and throw an exception with a helpful message for the developer.

Transactions aren't allowed to span multiple schema domains because after the partition, those transactions will no longer be able to guarantee consistency.

MySQL transactions guarantee consistency across tables within a database, but partitioned schema domains will be in different database clusters.

Moving Data without Downtime

Now that GitHub has virtually isolated schema domains, they can physically move their schema domains to separate database clusters.

In order to do this on the fly, GitHub uses Vitess. Vitess is an open source database clustering system for MySQL that was originally developed at YouTube.

Vitess was serving all YouTube database traffic from 2011 to 2019, so it's battle-tested.

GitHub uses Vitess' vertical sharding feature to move sets of tables together in production without downtime.

To do that, GitHub uses Vitess' VTGate proxies as the endpoint for applications to connect to instead of direct connections to MySQL.

Vitess handles the rest.

For more details, you can read the full blog post [here](#).

LinkedIn's journey of scaling HDFS to 1 Exabyte

LinkedIn is the world's largest professional social networking site with 800 million users from over 200 countries.

In order to run analytics workloads on all the data generated by these users, LinkedIn relies on Hadoop.

More specifically, they store all this data on the Hadoop Distributed File System (HDFS).

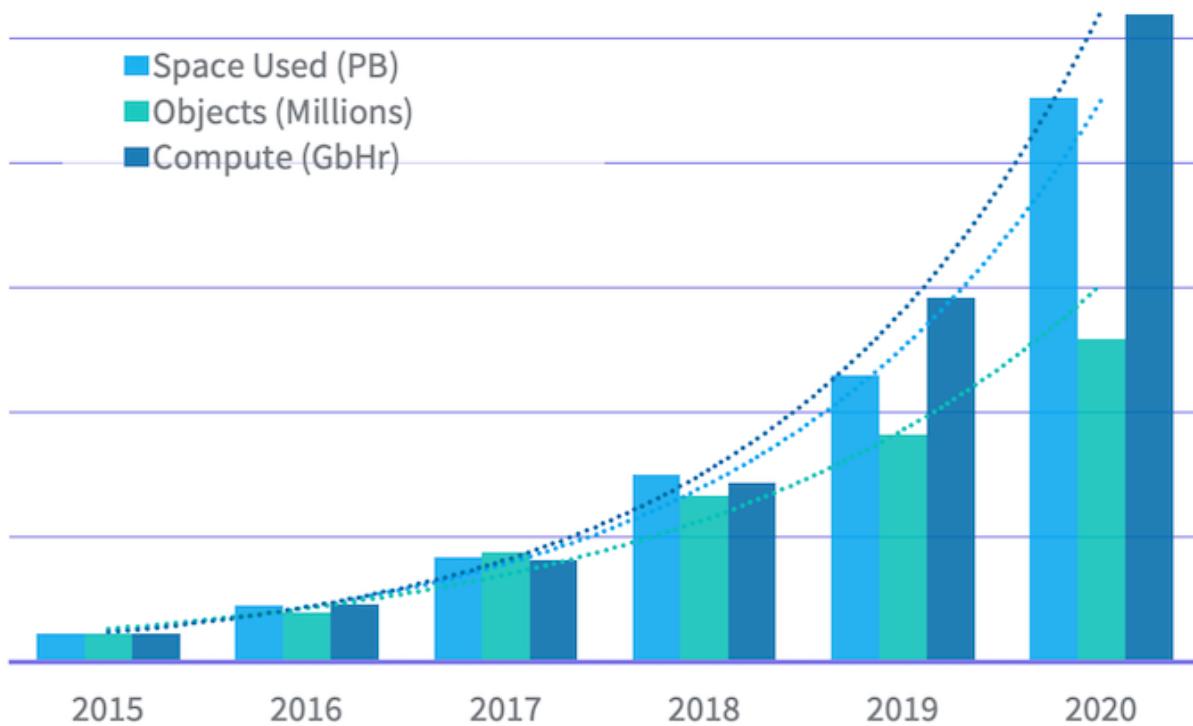
HDFS was based on Google File System (GFS), and you can read our article on GFS [here](#).

Over the last 5 years, LinkedIn's analytics infrastructure has grown *exponentially*, doubling every year in data size and compute workloads.

In 2021, they reached a milestone by storing 1 *exabyte* of data (1 million terabytes) across all their Hadoop clusters.

The largest Hadoop cluster stores 500 petabytes of data and needs over 10,000 nodes in the cluster. This makes it one of the largest (if not the largest) Hadoop cluster in the industry.

Despite the massive scale, the average latency for RPCs (remote procedure calls) to the cluster is under 10 milliseconds.



In 2015, the largest Hadoop cluster stored just over 20 petabytes of data.

It took just 5 years for that cluster to grow to over 200 petabytes of data.

In the article, LinkedIn engineers talk about some of the steps they took to ensure that HDFS could scale to 500 terabytes.

Replicating NameNodes

With HDFS, the file system metadata is decoupled from the data.

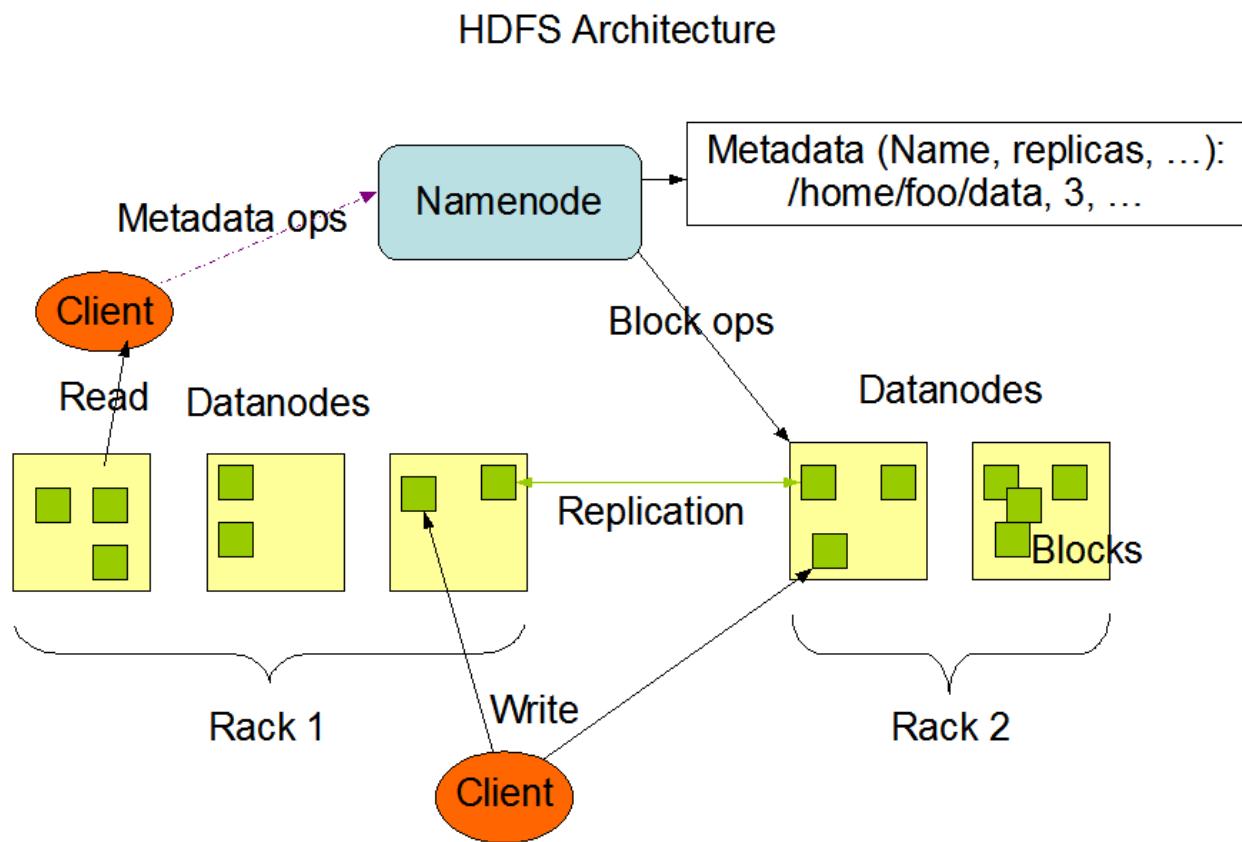
An HDFS cluster consists of two types of servers: a NameNode server and a bunch of DataNode servers.

The DataNodes are responsible for storing the actual data in HDFS. When clients are reading/writing data to the distributed file system, they are communicating directly with the DataNodes.

The NameNode server keeps track of all the file system metadata like the directory tree, file-block mappings (Hadoop breaks up files into 128 megabyte units called data blocks), which block is stored on which DataNodes, etc.

The NameNode also helps coordinate all the action in HDFS.

A client will first ask NameNode for the location of a certain file. NameNode will respond with the DataNode that contains that file and the client can then read/write their data directly to that DataNode server.



If your NameNode server goes down, then that's no bueno. Your entire Hadoop cluster will be down as the NameNode is a single point of failure.

Also, when you're operating at the scale of hundreds of petabytes in your cluster, restarting NameNode can take *more than an hour*. During this time, all jobs on the cluster must be suspended.

This is a big problem if you have a 500 petabyte cluster.

When you have clusters that large, it becomes *extremely* expensive to have downtime since a massive amount of processes at the organization rely on that cluster (that cluster accounts for *half* of all the data at LinkedIn).

Additionally, upgrading the cluster also becomes an issue since you have to restart the NameNode. This results in hours of additional downtime.

Fortunately, Hadoop 2 introduced a [High Availability feature](#) to solve this issue. With this feature, you can have multiple replicated NameNode servers.

The way it works is that you have a single *Active* NameNode that receives all the client's requests.

The Active NameNode will publish its transactions into a Journal Service (LinkedIn uses [Quorum Journal Manager](#) for this) and the Standby NameNode servers will consume those transactions and update their namespace state accordingly.

This keeps them up-to-date so they can take over in case the Active NameNode server fails.

LinkedIn uses IP failover to make failovers seamless. Clients communicate to the Active NameNode server using the same Virtual IP address irrespective of which physical NameNode server is assigned as the Active NameNode. A transition between NameNode servers will happen transparently to the clients.

Now, performing rolling updates is also much easier.

First, one of the Standby NameNodes is upgraded with the new software and restarted.

Then, the Active NameNode fails over to the upgraded standby and is subsequently upgraded and restarted.

After, the DataNodes can also be restarted with the new software. The DataNode restarts are done in batches, so that at least one replica of a piece of data remains online at all times.

Java Tuning

Previously, we talked about how the NameNode server will keep track of all the file system metadata.

The NameNode server keeps all of this file system metadata *in RAM* for low latency access.

As the filesystem grows, the namespace will also grow proportionally.

This adds the requirement for periodic increases of the Java heap size on the NameNode server (Hadoop is written in Java).

LinkedIn's largest NameNode server is set to use a 380 gigabyte heap to maintain the namespace for 1.1 billion objects.

Maintaining such a large heap requires elaborate tuning in order to provide high performance.

The Java heap is generally divided into two spaces: Young generation and Tenured (Old) generation.

An object will first start in the young generation, and as it survives garbage collection events, it will get promoted to eventually end up in the old generation.

As the workload on the NameNode increases, it generates more temporary objects in the young generation space.

The growth of the namespace increases the old generation.

LinkedIn engineers try to keep the storage ratio between the young and old generations at around 1:4.

By keeping the young and Old spaces appropriately sized, LinkedIn can completely avoid full garbage collection events (where both the Young and Old generations are collected), which would result in a many-minutes-long outage in the NameNode.

Non-Fair Locking

NameNode is a highly multithreaded application and it uses a global read-write lock to control concurrency.

The write lock is exclusive (only one thread can hold it and write) while the read lock is shared, allowing multiple reader threads to run while holding it.

Locks in java support two modes

- Fair - Locks are acquired in FIFO order
- Non-fair - Locks can be acquired out of order

With fair locking, the NameNode server frequently ends up in situations where writer threads block reader threads (where the readers could be running in parallel).

Non-fair mode, on the other hand, allows reader threads to go ahead of the writers.

This results in a substantial improvement in overall NameNode performance, especially since the workload is substantially skewed towards read requests.

Other Optimizations

Satellite Clusters

HDFS is optimized for maintaining large files and providing high throughput for sequential reads and writes.

As stated before, HDFS splits up files into blocks and then stores the blocks on the various DataNode servers.

Each block is set to a default size of 128 megabytes (LinkedIn has configured their cluster to 512 megabytes).

If lots of small files (the file size is less than the block size) are stored on the HDFS cluster, this can create issues by disproportionately inflating the metadata size compared to the aggregate size of the data.

Since all metadata is stored in the NameNode server's RAM, this becomes a scalability limit and a performance bottleneck.

In order to ease these limits, LinkedIn created Satellite HDFS Clusters that handled storing these smaller files.

You can read the details on how they split off the data from the main cluster to the satellite clusters in the [article](#).

Consistent Reads from Standby NameNodes

The main limiting factor for HDFS scalability eventually becomes the performance of the NameNode server.

However, LinkedIn is using the High Availability feature, so they have multiple NameNode servers (one in Active mode and the others in Standby state).

This creates an opportunity for reading metadata from Standby NameNodes instead of the Active NameNode.

Then, the Active NameNode can just be responsible for serving write requests for namespace updates.

In order to implement this, LinkedIn details the consistency model they used to ensure highly-consistent reads from the Standby NameNodes.

Read the full details in the [article](#).

Building a Static Analysis tool at Slack

Nicholas Lin and David Frankel were two interns on Slack's Product Security team, and they worked on building a static code analysis tool for Slack's codebase.

In case you're unaware, Slack is a chat tool catering towards businesses.

Here's a summary

Static Code Analysis tools inspect your source code (without executing it) and identify potential errors and security vulnerabilities.

Giving engineers static analysis tools can immensely help developer productivity and make the codebase much more secure.

Slack's codebase is largely written in the Hack programming language.

Hack was developed at Facebook and is a typed dialect of PHP (Hack allows both dynamic and static typing, so its type system is classified as [gradually typed](#)).

There were no static analysis tools available for Hack, so Nicholas and David set out to build one.

Building a static analysis tool from scratch would be too complex, so they decided to extend an existing open source static analysis tool, [Semgrep](#).

Semgrep was already in use at Slack to scan code in 6 different languages, and there's already infrastructure in place to integrate Semgrep into the CI/CD pipeline.

In order to add Hack functionality to Semgrep, they needed to answer two questions

1. What are the grammar rules for the Hack language?
2. How can Semgrep understand these grammar rules?

Developing a grammar for Hack

Programming languages have a structure to them that is known as the [grammar](#).

The main notation used to represent grammars is the Backus-Naur Form (BNF).

Here's an example of a very simple grammar that can recognize arithmetic expressions.

```
<exp> ::= <exp> "+" <exp>
<exp> ::= <exp> "*" <exp>
<exp> ::= "(" <exp> ")"
<exp> ::= "a"
<exp> ::= "b"
<exp> ::= "c"
```

A programming language's grammar will let you transform the program from a series of ASCII characters (words, spaces, etc.) into a [concrete syntax tree \(also known as a parse tree\)](#).

The concrete syntax tree (CST) is an exact visual representation of the parsed source code based on the grammar.

```
≡ script.hack
1     function main(): void { print "wyd, world\\n"; }
2

≡ script.exp U X

≡ script.exp
1   (script [0, 0] - [1, 0]
2     (function_declarati[...]
```

Example tree-sitter CST (bottom) generated from Hack source (top)

So, Nicholas and David incrementally wrote the grammar rules for the Hack programming language.

They tested their grammar by using a library called Tree-sitter.

Tree-sitter can take your grammar rules and then generate a language parser from them.

Nicholas and David used Tree-sitter to create a Hack parser using their grammar.

Then, they tested that parser by using it to convert some of Slack's source code into a CST.

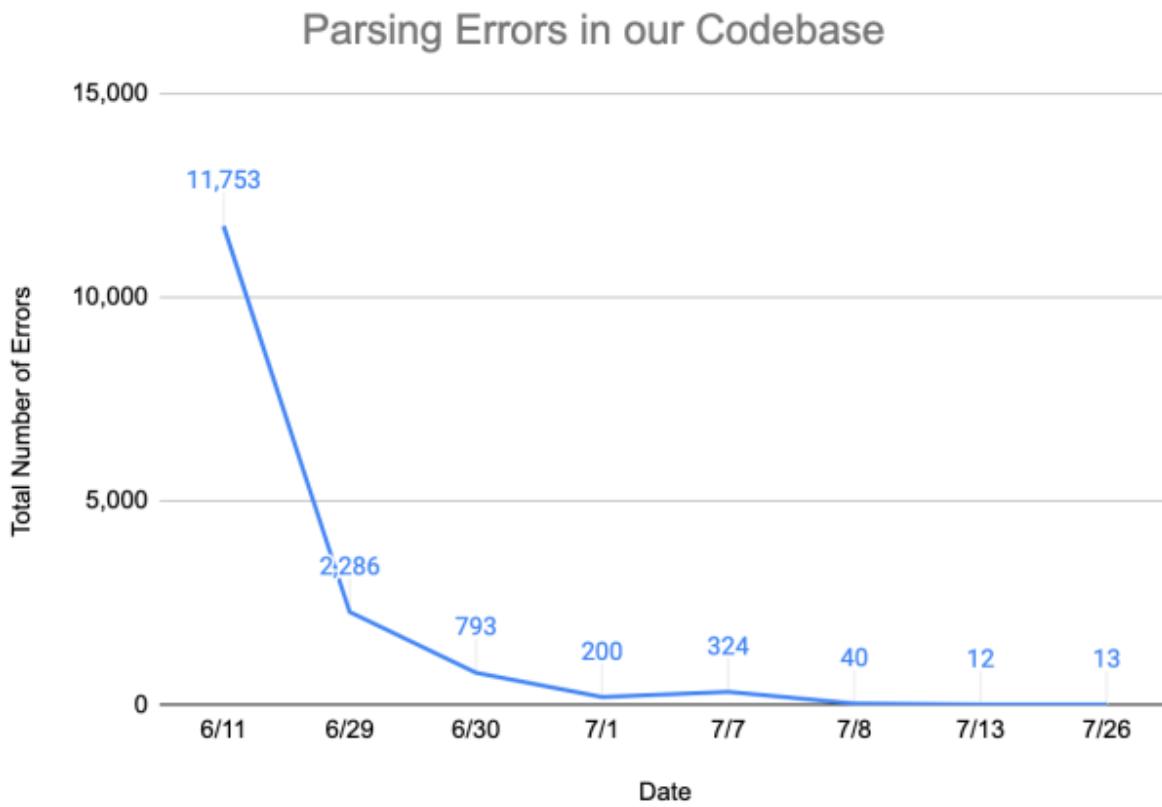
From this conversion, they can measure the *parse rate*, which is the proportion of the source code that could be properly parsed to construct a CST.

The higher the parse rate, the more code your parser was able to understand to construct the CST.

Since the parser is based on the grammar rules, having a very high parse rate means that your grammar is very expressive and covers the programming language well.

Nicholas and Dave were eventually able to develop a grammar that achieved a parse rate of greater than 99.999%.

Of the 5 million lines of Hack code they tested, there were less than 15 lines of unparsable code.



The Hack grammar they developed is open source. You can view it [here](#).

Teaching Semgrep the Grammar

Semgrep (the static analysis tool) uses an abstract syntax tree to understand the source code and find bugs/vulnerabilities.

While the CST is an exact representation of your code, the abstract syntax tree (AST) focuses on the essential information.

An AST focuses on the structure of your code, and represents it in a hierarchical data structure useful for analysis.

You can read about the differences between an AST and a CST [here](#).

Semgrep converts your source code (in Go, Java, JavaScript, Python, Ruby, etc.) into a language-agnostic AST.

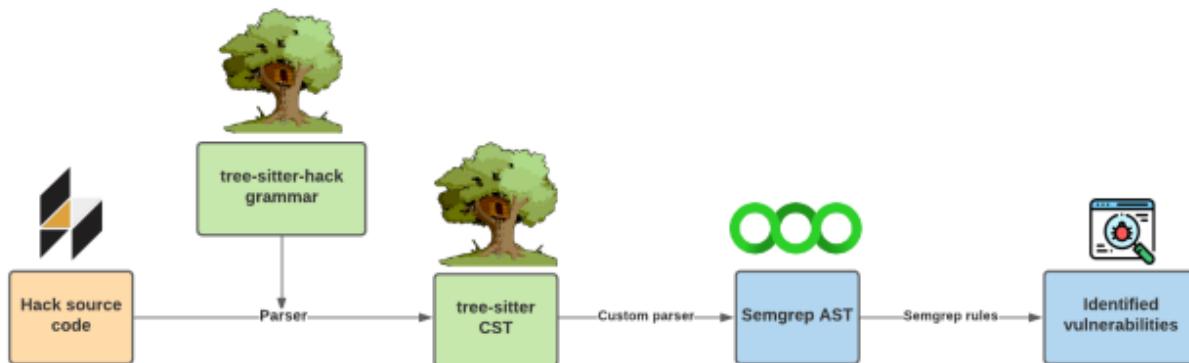
Then, it looks through a list of rules that check the Semgrep AST for bugs and vulnerabilities.

This makes Semgrep highly extensible, as it's loosely coupled with the programming language.

In order to map the tree-sitter CST to the Semgrep AST, Nicholas and David wrote a custom parser file in OCaml (Semgrep-core is written in the OCaml programming language).

Then, they plugged this file into Semgrep and used the static analysis capabilities with Hack code.

For more details, read the full article [here](#).



How Facebook Encodes Videos

Hundreds of millions of videos are uploaded to Facebook every day.

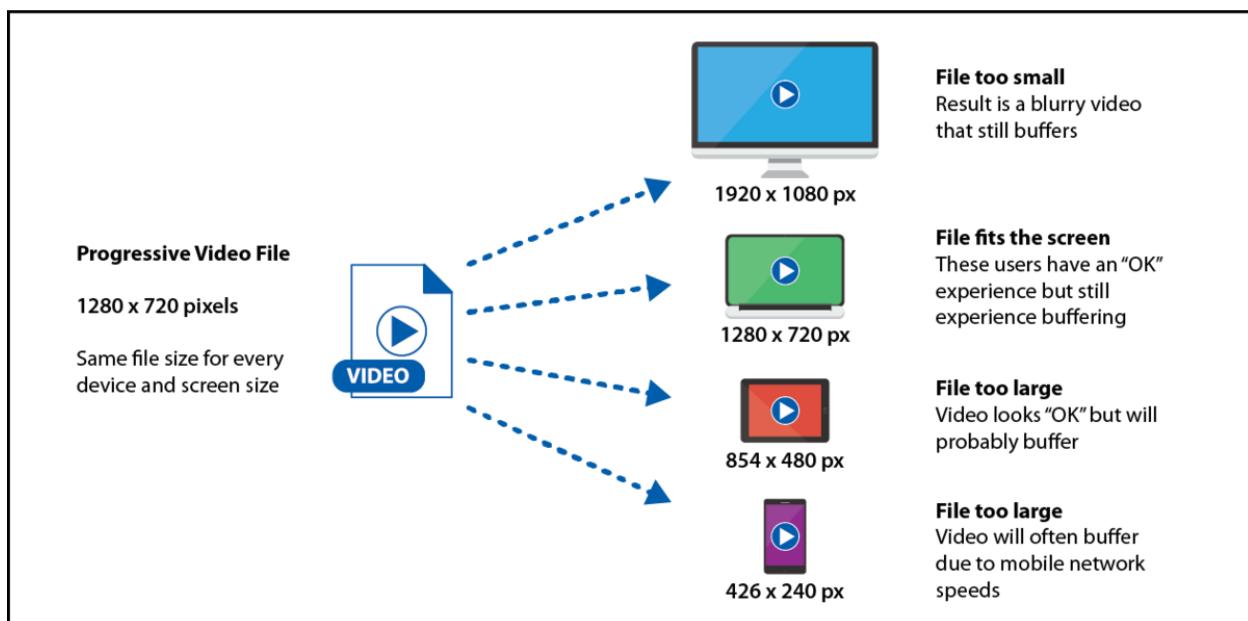
In order to deliver these videos with high quality and little buffering, Facebook uses a variety of video codecs to compress and decompress videos. They also use Adaptive Bitrate Streaming (ABR).

We'll first give a bit of background information on what ABR and video codecs are. Then, we'll talk about the process at Facebook.

Progressive Streaming vs. Adaptive Bitrate Streaming

Progressive Streaming is where a single video file is being streamed over the internet to the client.

The video will automatically expand or contract to fit the screen you are playing it on, but regardless of the device, the video file size will always be the same.



There are numerous issues with progressive streaming.

- Quality Issue - Your users will have different screen sizes, so the video will be stretched/pixelated if their screen resolution is different from the video's resolution.
- Buffering - Users who have a poor internet connection will be downloading the same file as users who have a fast internet connection, so they (slow-download users) will experience much more buffering.

Adaptive Bitrate Streaming is where the video provider creates different videos for each of the screen sizes that he wants to target.

He can encode the video into multiple resolutions (480p, 720p, 1080p) so that users with slow internet connections can stream a smaller video file than users with fast internet connections.

The player client can detect the user's bandwidth and CPU capacity in real time and switch between streaming the different encodings depending on available resources.

You can read more about Adaptive Bitrate Streaming [here](#).

Video Codec

A video codec compresses and decompresses digital video files.

Transmitting uncompressed video data over a network is impractical due to the size (tens to hundreds of gigabytes).

Video codecs solve this problem by compressing video data and encoding it in a format that can later be decoded and played back.

Examples of common codecs include H264 (AVC), MPEG-1, VP9, etc.

The various codecs have different trade-offs between compression efficiency, visual quality, and how much computing power is needed.

More advanced codecs like VP9 provide better compression performance over older codecs like H264, but they also consume more computing power.

You can read more about video codecs [here](#).

Facebook's Process for Encoding Videos

So, you upload a video of your dog to Facebook. What happens next?

Once the video is uploaded, the first step is to encode the video into multiple resolutions (360p, 480p, 720p, 1080p, etc.)

Next, Facebook's video encoding system will try to further improve the viewing experience by using advanced codecs such as H264 and VP9.

The encoding job requests are each assigned a priority value, and then put into a priority queue.

A specialized encoding compute pool then handles the job.

Now, the Facebook web app (or mobile app) and Facebook backend can coordinate to stream the highest-quality video file with the least buffering to people who watch your video.

A key question Facebook has to deal with here revolves around *how they should assign priority values to jobs?*

The goal is to maximize everyone's video experience by quickly applying more compute-intensive codecs to the videos that are watched the most.

Let's say Cristiano Ronaldo uploaded a video of his dog at the same time that you uploaded your video.

There's probably going to be a lot more viewers for Ronaldo's video compared to yours so Facebook will want to prioritize encoding for Ronaldo's video (and give those users a better experience).

They'll also want to use more computationally-expensive codecs (that result in better compression ratios and quality) for Ronaldo.

The Benefit-Cost Model

Facebook's solution for assigning priorities is the Benefit-Cost model.

It relies on two metrics: Benefit and Cost.

The encoding job's priority is then calculated by taking Benefit and dividing it by Cost.

1. **Benefit** = (relative compression efficiency of the encoding family at fixed quality) * (effective predicted watch time)
2. **Cost** = normalized compute cost of the missing encodings in the family
3. **Priority** = Benefit/Cost

Benefit

The benefit metric attempts to quantify how much benefit Facebook users will get from advanced encodings.

It's calculated by multiplying *relative compression efficiency* * *effective predicted watch time*.

The effective predicted watch time is an estimate of the total watch time that a video will be watched in the near future across all of its audience.

Facebook uses a sophisticated ML model to predict the watch time. They talk about how they created the model (and the parameters involved) in the article.

The relative compression efficiency is a measure of how much a user benefits from the codec's efficiency.

It's based on a metric called the Minutes of Video at High Quality per GB (MVHQ) which is a measure of how many minutes of high-quality video can you stream per gigabyte of data.

Facebook compares the MVHQ of different encodings to find the relative compression efficiency.

Cost

This is a measure of the amount of logical computing cycles needed to make the encoding family (consisting of all the different resolutions) deliverable.

Some jobs may require more resolutions than others before they're considered deliverable.

As stated before, Facebook divides Benefit / Cost to get the priority for a video encoding job.

After encoding, Facebook's backend will store all the various video files and communicate with the frontend to stream the optimal video file for each user.

For more details, read the full article [here](#).

How Uber Migrated their Financial Transaction Database from DynamoDB to Docstore

In order to keep track of their financial transaction data, Uber built an immutable ledger-style database called LedgerStore.

LedgerStore's storage backend was AWS DynamoDB but Uber decided to migrate away because it was becoming expensive.

They switched the storage backend to [Docstore](#), a general-purpose, [multimodal](#) database that was developed internally at Uber.

The project resulted in

- \$6 million of yearly savings for Uber
- Fewer external dependencies
- Technology Consolidation (since Docstore is an Uber product, many other services inside Uber also use it)
- Latency improvements

Uber was able to do this without a single production incident and not a single data inconsistency in the 250 billion unique records that were migrated from DynamoDB to Docstore.

Piyush Patel, Jaydeepkumar Chovatia and Kaushik Devarajaiah wrote a [great article](#) on the migration, and we'll be giving a summary below.

Here's a summary

Uber moves millions of people around the world and delivers tens of millions of food orders daily. This generates a massive amount of financial transactions that need to be stored.

To do this, Uber uses LedgerStore, an append-only, ledger-style distributed database.

A datastore in LedgerStore is a collection of tables and each table contains a set of records modeled as documents.

Here's an example of a LedgerStore table schema

```
CREATE TABLE ORDERS (
    order_uuid STRING,
    business_ts INT64,
    order_blob BLOB,
    city_id INT32,
    amount DOUBLE,
) PRIMARY KEY(order_uuid) ROOT LSG.SEALING(after=30m);

CREATE GLOBAL INDEX cities ON ORDERS (city_id) STORING (amount);
```

Tables can have one or many indexes and an index belongs to exactly one table.

Indexes in LedgerStore are strongly consistent, so when a write to the main table succeeds, all indexes are updated at the same time using a [2-phase commit](#).

LedgerStore also provides automatic data-tiering functionality.

The data is mostly read within a few weeks or months after being written. Because it's expensive to store data in hot databases like DynamoDB, Uber offloads the data to colder storage after a time period.

LedgerStore was designed to provide the following data integrity guarantees.

1. Individual records are immutable
2. Corrections are trackable
3. Unauthorized data changes and inconsistencies must be detected
4. Queries are reproducible a bounded time after write

In order to provide these guarantees, Ledger Store created the concept of Sealing.

Sealing

Sealing is the process of closing a past time range of data for changes and maintaining signatures of the data within that sealed time range.

After a sealing window is closed, signed and sealed, no further updates to it will be permitted.

If you need to correct data in an already-sealed time range, LedgerStore uses Revisions, which we'll discuss below.

Because there are no updates, any query that only reads data from a sealed time range is guaranteed to be reproducible.

Revisions

If you need to correct data in already-sealed time ranges, LedgerStore uses the notion of Revisions.

A revision is a table-level entity consisting of all sealed record corrections and the associated business justifications. All records, both corrected and the original, are maintained to allow reproducible queries.

Choosing Docstore

LedgerStore was designed to abstract away the underlying storage technology so that switching technologies could be done if the business need arose.

As the database scaled, using AWS DynamoDB as a storage backend became extremely expensive.

Additionally, having different backend databases in the tech stack created fragmentation and made it difficult to operate.

The requirements for the storage backend were

- High availability - 99.99% availability guarantees
- Can be easily scaled horizontally

- [Change Data Capture](#) (a.k.a streaming)
- Secondary Indexes
- A flexible data model

[Docstore](#), Uber's homegrown database, was a perfect match for those requirements.

The only issue was that Docstore didn't have Change Data Capture (streaming) functionality.

Uber wanted streaming functionality because reading data from a stream of updates is more efficient than reading from the table directly.

You don't have to perform table scans or range reads spawning a large number of rows. Also, the stream data can be stored in cheaper, commodity hardware.

The stream data can be stored in a system like Apache Kafka, which is optimized for stream reading.

Uber solved this issue by building a streaming framework for Docstore called Flux. Read the article for more details on Flux.

DynamoDB to Docstore Migration

When migrating from DynamoDB to Docstore, Uber had several objectives they wanted

- Zero stakeholder involvement - clients who rely on LedgerStore should not be involved or exposed to the migration. They shouldn't have to change any code.
- High Availability - no downtime during migration
- Maintaining 100% Read-Your-Writes data consistency
- Maintaining pre-existing performance SLOs, such as latency
- Ability to switch back, in case of emergency

Historical Data Migration

One part of the migration was moving all the historical data from DynamoDB to Docstore in real time.

The data consisted of more than 250 billion unique records and was 300 terabytes of data in total.

Engineers did this by breaking the historical data down into subsets and then processing them individually via checkpointing.

LedgerStore already supported cold storage offloading (part of automatic data-tiering discussed earlier) and the offloading worked at a sealing window granularity.

Therefore, the data is already broken down into subsets, where each subset is an individual sealing window.

Engineers built a backfill framework to process individual sealing windows and maintain a checkpoint of them.

The framework is multi-threaded, spawning multiple workers where each worker is processing a distinct sealing window.

As a result, the framework is capable of processing 1 million messages per second. Backfilling all the historical data was able to be done in a couple of weeks.

Online Traffic Redirection

The second part of the migration is online traffic redirection.

To do this, engineers configured LedgerStore to talk to 2 different databases (DynamoDB and Docstore), with each database assuming a primary or secondary role depending on the phase of the migration.

The goal was to keep the 2 databases consistent at any phase of the migration so that rollback and forward were possible.

The online traffic redirection was divided into 4 phases

1. Shadow Writes - engineers added a shadow writer module in LedgerStore's write path to insert incoming data into the secondary database along with the primary. The secondary database writes happened asynchronously, to keep write latency low. A [two-phase commit protocol](#) was used to track the asynchronous writes.
2. Dual Read and Merge - To guarantee 100% consistency and 99.99% availability, a new module called Dual Read and Merge was added in the read path for LedgerStore. This module served read requests by reading from both databases, merging the results, and then returning them to the client.
3. Swap Databases - In this phase, engineers compared both the Docstore database with the DynamoDB database to ensure it matched. They were able to validate 250 billion unique records through a Spark job within a single week. After validation, they swapped the databases and promoted Docstore as the primary and DynamoDB as the secondary. After Docstore is promoted to primary, they slowly stopped traffic to DynamoDB. First, they gradually removed reads from DynamoDB and served them out of Docstore.
4. Final Cutover - Once reads were fully served out of Docstore, it was time to stop the shadow writes to DynamoDB. Afterwards, engineers backed up the DynamoDB database and finally decommissioned it.

[Read the full article for more details](#)

An Introduction to Big Data Architectures

Microsoft has a great introduction to Big Data Architectures in their Azure [docs](#).

Here's a summary.

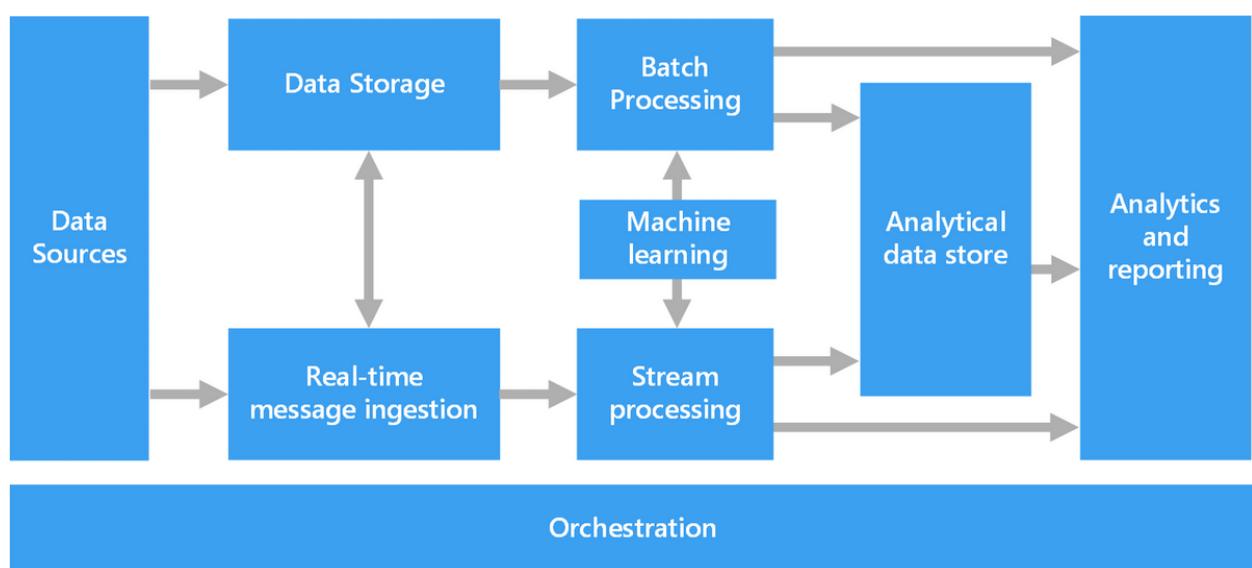
When you're dealing with massive amounts of data (hundreds of terabytes or petabytes), then the traditional ways of dealing with data start to break down.

You'll need to use a distributed system, and you'll have to orchestrate the different components to suit your workload.

Typically, big data solutions involve one or more of the following types of workloads.

- Batch processing of data in storage
- Real-time processing of data from a stream
- Interactive exploration of data
- Predictive analytics and machine learning

Components of a Big Data Architecture



Most big data architectures include some or all of the following components

- *Data Sources* - It's pretty dumb to build a distributed system for managing data if you don't have any data. So, all architectures will have at least one source that's creating data. This can be a web application, an IoT device, etc.
- *Data Storage* - Data for batch processing operations is typically stored in a distributed file store that can hold high volumes of large files in various formats. This kind of store is often called a [data lake](#).
- *Batch Processing* - Because the data sets are so large, often a big data solution must process data files using long-running batch jobs to filter, aggregate, and prepare the data for analysis. [Map Reduce](#) is a very popular way of running these batch jobs on distributed data.
- *Real-time Message Ingestion* - If the solution includes real-time sources, the architecture must include a way to capture and store real-time messages for stream processing. This portion of a streaming architecture is often referred to as stream buffering.
- *Stream Processing* - After capturing real-time messages, the solution must process them by filtering, aggregating, and otherwise preparing the data for analysis. [Apache Storm](#) is a popular open source technology that can handle this for you.
- *Analytical Data Store* - Many big data solutions prepare data for analysis and then serve the processed data in a structured format that can be queried using analytical tools. This can be done with a relational [data warehouse](#) (commonly used for BI solutions) or through NoSQL technology like [HBase](#) or [Hive](#).
- *Orchestration* - Big data solutions consist of repeated data processing operations that transform source data, move data between multiple sources and sinks, load the processed data into an analytical data store, or push the

results straight to a report or dashboard. You can use something like [Apache Oozie](#) to orchestrate these jobs.

When you're working with a large data set, analytical queries will often require batch processing. You'll have to use something like MapReduce.

This means that getting an answer to your query can take *hours*, as you have to wait for the batch job to finish.

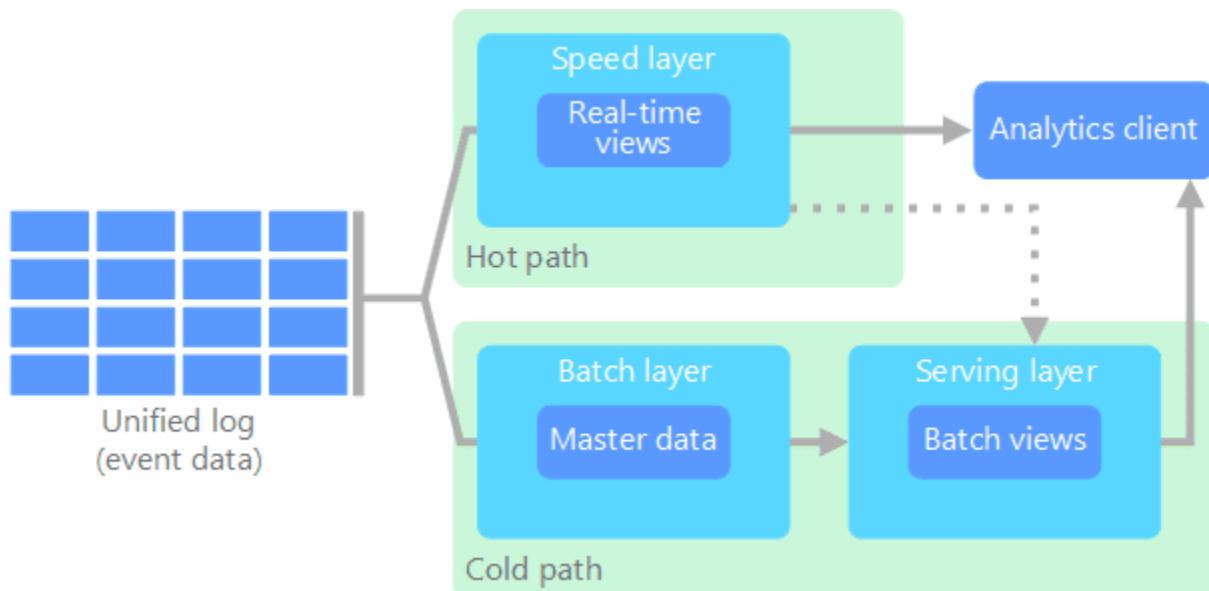
The issue is that this means you won't get real time results to your queries. You'll always get an answer that is a few hours old.

This is a problem that comes up frequently with big data architectures.

The ideal scenario is where you can get some results in real time (perhaps with some loss of accuracy) and combine these results with the results from the batch job.

The Lambda Architecture is a solution to this issue.

Lambda Architecture



The Lambda Architecture solves this by creating two paths for data flow: the *cold* path and the *hot* path.

The cold path is also known as the batch layer. It stores all of the historical data in raw form and performs batch processing on the data.

The raw data in the batch layer is immutable. The incoming data is always appended to the existing data, and the previous data is never overwritten.

The cold path has a high latency when answering analytical queries. This is because the batch layer aims at perfect accuracy by processing all available data when generating views.

The hot path is also known as the speed layer, and it analyzes the incoming data in real time. The speed layer's views may not be as accurate or complete as the batch layer, but they're available almost immediately after the data is received.

The speed layer is responsible for filling the gap caused by the batch layer's lag and provides views for the most recent data.

The hot and cold paths converge at the serving layer. The serving layer indexes the batch view for efficient querying and incorporates incremental updates from the speed layer based on the most recent data.

With this solution, you can run analytical queries on your datasets and get up-to-date answers.

A drawback to the lambda architecture is the complexity. Processing logic appears in two different places (the hot and cold paths) and they use different frameworks.

The Kappa Architecture is meant to be a solution to this, where all your data flows through a single path, using a stream processing engine.

You can read more about the Kappa Architecture [here](#).

How Stripe uses Similarity Clustering to catch Fraud Rings

Stripe is one of the world's largest payment processors.

The company's main product is the Stripe Payments API, which developers can use to easily embed payment functionality into their applications.

Due to Stripe's scale, they're a big target for payments fraud and cybercrime.

Andrew Tausz is part of the Risk Intelligence team at Stripe, and he wrote a great [blog post](#) on how Stripe uses similarity clustering to catch fraud rings.

Merchant Fraud at Stripe

One of the most common types of fraud that Stripe faces is merchant fraud, where a scammer will create a website that advertises fraudulent products or services (and uses Stripe to process payments).

An example might be if a scammer creates a website that sells electronic goods at a highly discounted price. After a customer pays him for the good, he pockets the money and doesn't send the customer the promised good.

The customer will end up issuing a chargeback through their credit card, which will eventually get paid back by Stripe.

Stripe will then attempt to debit the account of the scammer, but if they're unable to (the scammer transferred out all his money) then Stripe will have to eat the losses.

After a fraudster gets caught by Stripe, his account will be disabled. But, it's quite likely that he'll try to continue the scam by creating a new Stripe account.

One way Stripe can reduce fraud is by catching these repeat fraudsters through *similarity clustering*.

Using Similarity Clustering to Reduce Merchant Fraud

When a scammer creates a new Stripe account (after getting caught on his previous account), he'll probably reuse some information and attributes from his previous account.

Certain information is easy to fabricate, like your name or date of birth. But, other attributes are more difficult. For example, it takes significant effort to obtain a new bank account.

Therefore, Stripe has found that linking accounts together via shared attributes is quite effective at catching obvious fraud attempts.

Switching from Heuristics-based to an ML model

In order to link accounts together, Stripe relies on a *similarity score*.

They take two accounts and then assign them a similarity score based on the number of shared attributes the accounts have.

Some shared attributes are weighed more heavily than others. Two Stripe accounts who share dates of birth should have a lower similarity score than two accounts who share a bank account.

Previously, Stripe relied on a heuristic based system where the weightings were hand-constructed (based on guess and check).

Stripe decided to switch by training a machine learning model to handle this task.

Now, they can automatically retrain the model over time as they obtain more data and improve in accuracy, adapt to new fraud trends, and learn the signatures of particular adversarial groups.

Building the ML Model

To build the model, Stripe followed a [supervised learning](#) approach.

The approach Stripe took to build the model is [Similarity Learning](#), where the objective is to learn a similarity function that can measure how similar two objects are.

Similarity learning is used extensively in ranking, recommendation systems, face/voice verification, and fraud detection.

They already had a massive dataset of fraud rings and clusters of fraudulent accounts based on prior work from their risk underwriting team.

Stripe cleaned that into a dataset consisting of pairs of accounts along with a label for each pair indicating whether or not the two accounts belong to the same cluster.

Now that they had the dataset, Stripe had to generate [features](#) that the model could use to compare the pair of accounts.

Creating a Stripe account requires quite a bit of data, so Stripe had a large feature set they could utilize.

Examples of features chosen include the account's email domain, overlap in credit card numbers used for both accounts, measure of text similarity, and more.

Using gradient-boosted decision trees

Due to the huge range of features, Stripe decided to go with [gradient-boosted decision trees \(GBDTs\)](#) to represent their similarity model.

Stripe found that GBDTs strike the right balance between being easy to train, having strong predictive power, and being robust despite variations in the data.

GBDTs are also straightforward to fine-tune and have well-understood properties.

The implementation of GBDTs that Stripe used was [XGBoost](#).

Stripe chose XGBoost models because of their great performance and also because Stripe already had well-developed infrastructure to train and support them.

Stripe has an internal API called Railyard that handles training ML models in a scalable and maintainable way.

You can read more about Railyard and its architecture [here](#).

Prediction Use

After, Stripe began to use their model to predict fraudulent activity.

Since this model operates on pairs of Stripe accounts, it's not possible to feed it all pairs of accounts and compute similarity scores across all pairs (there's too many combinations).

Instead, Stripe uses some heuristics to identify suspicious accounts and prune the set of candidates to a reasonable number.

Then, they use their ML models to generate similarity scores between the accounts.

After, they compute the connected components on the resulting graph to get a final output of high-fidelity account clusters that can be analyzed, processed or manually inspected.

If a cluster contains a large amount of known fraudulent accounts, then a risk analyst may want to further investigate the remaining accounts in that cluster.

You can read more details in the full article [here](#).

Evolving LinkedIn's Analytics Tech Stack

Steven Chuang, Qinyu Yue, Aaravind Rao and Srihari Duddukuru are engineers at LinkedIn. They published an interesting [blog post](#) on transitioning LinkedIn's analytics stack from proprietary platforms to open source big data technologies.

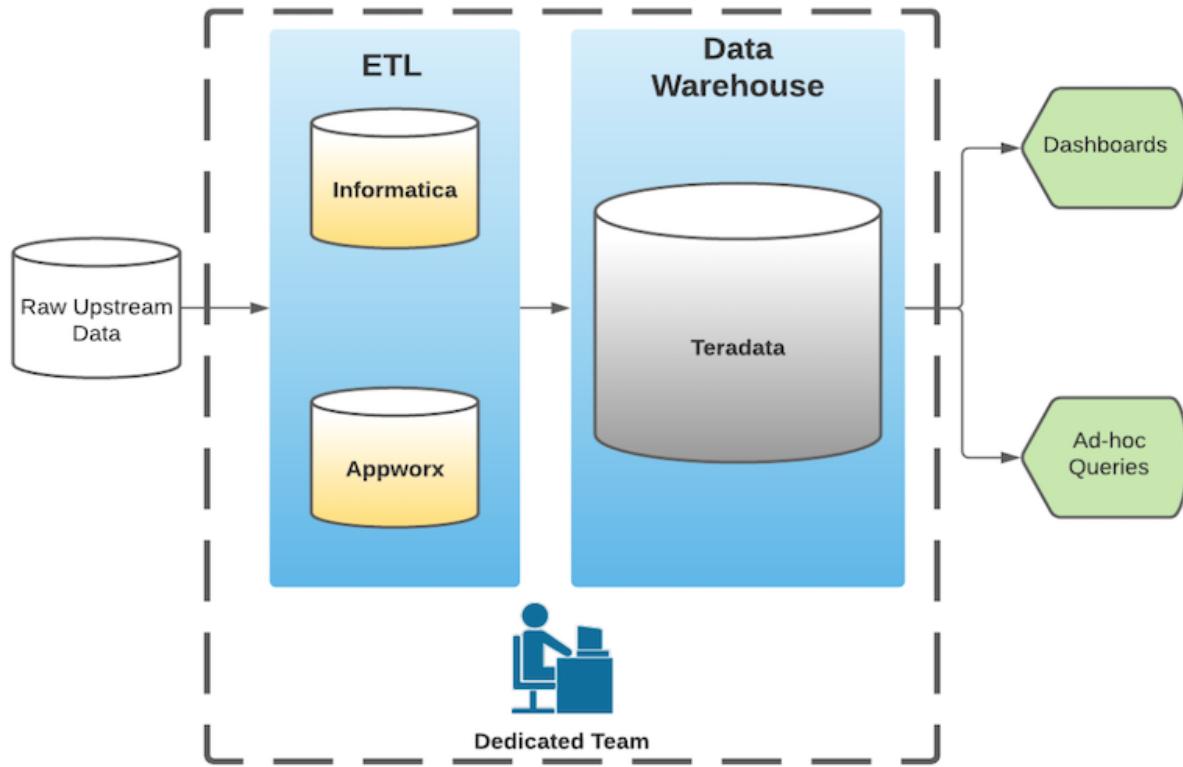
Here's a summary

During LinkedIn's early stages (early 2010s), they were growing extremely quickly. To keep up with this growth, they leveraged several third party proprietary platforms (3PP) in their analytics stack.

Using these proprietary platforms was far quicker than piecing together off-the-shelf products.

LinkedIn relied on Informatica and Appworx for ETL to a Data Warehouse built with Teradata.

ETL stands for Extract, Transfer, Load. It's the process of copying data from various sources (the different data producers) into a single destination system (usually a data warehouse) where it can more easily be consumed.



This stack served LinkedIn well for 6 years, but it had some disadvantages:

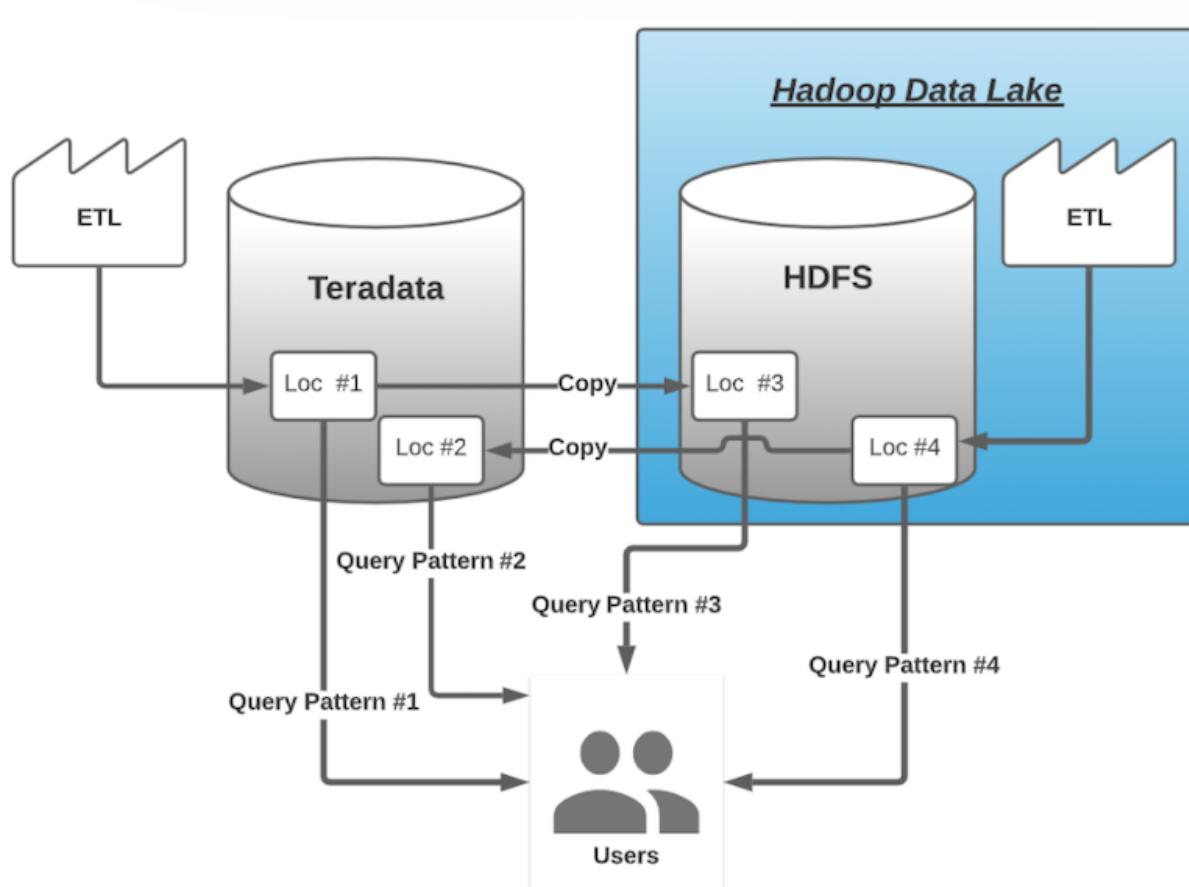
- Lack of freedom to evolve - Because of the closed nature of this system, they were limited in options for innovation. Also, integration with internal and open source systems was a challenge.
- Difficulty in scaling - Data pipeline development was limited to a small central team due to the limits of Informatica/Appworx licenses. This increasingly became a bottleneck for LinkedIn's rapid growth.

These disadvantages motivated LinkedIn engineers to develop a new [data lake](#) (data lakes let you contain raw data without having to structure it) on Hadoop in parallel.

You can read about how LinkedIn scaled Hadoop Distributed File System to 1 exabyte of data [here](#).

However, they did not have a clear transition process, and that led to them maintaining both the new system and the legacy system simultaneously.

Data was copied between the tech stacks, which resulted in double the maintenance cost and complexity.



Data Migration

To solve this issue, engineers decided to migrate all datasets to the new analytics stack with Hadoop.

In order to do this, the first step was to derive LinkedIn's data lineage.

Data lineage is the process of tracking data as it flows from data sources to consumption, including all the transformations the data underwent along the way.

Knowing this would enable engineers to plan the order of dataset migration, identify zero usage datasets (and delete them for workload reduction) and track the usage of the new vs. old system.

You can read exactly how LinkedIn handled the data lineage process in the full [article](#).

After data lineage, engineers used this information to plan major data model revisions.

They planned to consolidate 1424 datasets down to 450, effectively cutting ~70% of the datasets from their migration workload.

They also transformed data sets that were generated from OLTP workloads into a different model that was more suited for business analytics workloads.

The migration was done using various data pipelines and illustrated bottlenecks in LinkedIn's systems.

One bottleneck was poor read performance of the [Avro file format](#). Engineers migrated to [ORC](#) and consequently saw a read speed increase of ~10-1000x, along with a 25-50% improvement in compression ratio.

After the data transfer, depreciating the 1400+ datasets on the legacy system would be tedious and error prone if done manually, so engineers also built an automated system to handle this process.

They built a service to coordinate the deprecation where the service would identify dataset candidates for deletion (datasets with no dependencies and low usage) and then send emails to users of those datasets with news about the upcoming deprecation.

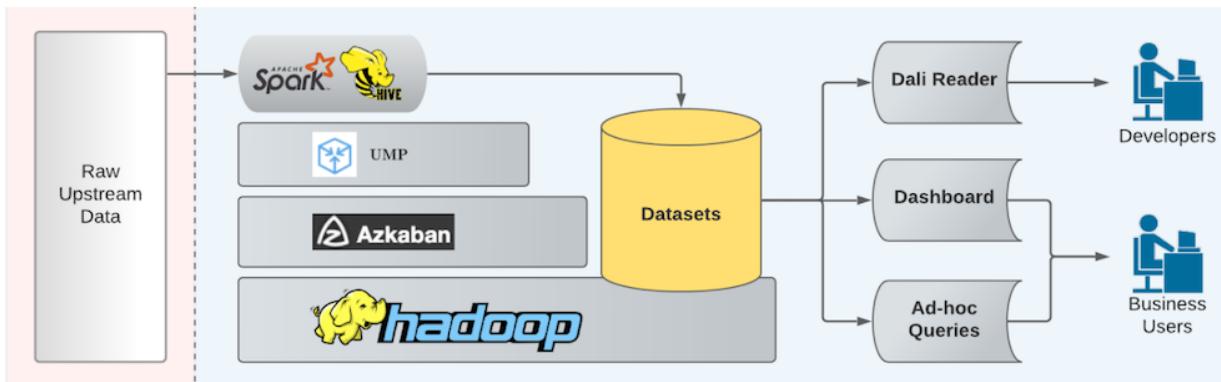
The service would also notify SREs to lock, archive and delete the dataset from the legacy system after a grace period.

The New System

The design of the new ecosystem was heavily influenced by the old ecosystem, and addressed the major pain points from the legacy tech stack.

- Democratization of data - The Hadoop ecosystem enabled data development and adoption by other teams at LinkedIn. Previously, only a central team could build data pipelines on the old system due to license limits with the proprietary platforms.

- Democratization of tech development with open source projects - All aspects of the new tech stack can be freely enhanced with open source or custom-built projects.
- Unification of tech stack - Simultaneously running 2 tech stacks showed the complexity and cost of maintaining redundant systems. Unifying the technology allowed for a big boost in efficiency.



The new tech stack has the following components

- Unified Metrics Pipeline - A unified platform where developers provide ETL scripts to create data pipelines.
- Azkaban - A distributed workflow scheduler that manages jobs on Hadoop.
- Dataset Readers - Datasets are stored on Hadoop Distributed File System and can be read in a variety of ways.
 - They can be read by DALI, an API developed to allow LinkedIn engineers to read data without worrying about its storage medium, path or format.
 - They can be read by various Dashboards and ad-hoc queries for business analytics.

For more details on LinkedIn's learnings and their process for the data (and user) migration, read the [full article](#).

Etsy's Journey to TypeScript

Salem Hilal is a software engineer on Etsy's Web Platform team. He wrote a great [blog post](#) on the steps Etsy took to adopt TypeScript.

Here's a summary

Etsy's codebase is a [monorepo](#) with over 17,000 JavaScript files, spanning many iterations of the site.

In order to improve the codebase, Etsy made the decision to adopt TypeScript, a superset of JavaScript with the optional addition of types. This means that any valid JavaScript code is valid TypeScript code, but TypeScript provides additional features on top of JS (the type system).

Based on [research at Microsoft](#), static type systems can heavily reduce the amount of bugs in a codebase. Microsoft researchers found that using TypeScript or [Flow](#) could have prevented 15% of the public bugs for JavaScript projects on Github.

Strategies for Adoption

There are countless different strategies for migrating to TypeScript.

For example, Airbnb [automated as much of their migration as possible](#) while other companies enable less-strict TypeScript across their projects, and add types to their code over time.

In order to determine their strategy, Etsy had to answer a few questions...

1. How strict do they want their flavor of TypeScript to be? - TypeScript can be [more or less “strict”](#) about checking the types in your codebase. A stricter configuration results in stronger guarantees of program correctness.
TypeScript is a superset of JavaScript, so if you wanted you could just rename all your .js files to .ts and still have valid TypeScript, but you would not get strong guarantees of program correctness.

2. How much of their codebase do they want to migrate? - TypeScript is designed to be easily adopted *incrementally* in existing JavaScript projects. Again, TypeScript is a superset of JavaScript, so all JavaScript code is valid TypeScript. Many companies opt to gradually incorporate TypeScript to help developers ramp up.
3. How specific do they want the types they write to be? - How accurately should a type fit the thing it's describing? For example, let's say you have a function that takes in the name of an HTML tag. Should the parameter's type be a string? Or, should you create a map of all the HTML tags and the parameter should be a key in that map (far more specific)?

Based on the previous questions, Etsy's adoption strategy looked like

1. Make TypeScript as strict as reasonably possible, and migrate the codebase file-by-file.
2. Add really good types and really good supporting documentation to all of the utilities, components, and tools that product developers use regularly.
3. Spend time teaching engineers about TypeScript, and enable TypeScript syntax team by team.

To elaborate more on each of these points...

Gradually Migrate to Strict TypeScript

Etsy wanted to set the compiler parameters for TypeScript to be as strict as possible.

The downside with this is that they would need *a lot* of type annotations.

They decided to approach the migration incrementally, and first focus on typing actively-developed areas of the site.

Files that had reliable types were given the .ts file extension while files that didn't kept the .js file extension.

Make sure Utilities and Tools have good TypeScript support

Before engineers started writing TypeScript, Etsy made sure that all of their tooling supported the language and that all of their core libraries had usable, well-defined types.

In terms of tooling, Etsy uses Babel and the plugin [babel-preset-typescript](#) that turns TypeScript into JavaScript. This allowed Etsy to continue to use their existing build infrastructure. To check types, they run the TypeScript compiler as part of their test suite.

Etsy makes heavy use of custom ESLint linting rules to maintain code quality.

They used the [TypeScript ESLint](#) project to get a handful of TypeScript specific linting rules.

Educate and Onboard Engineers Team by Team

The *biggest* hurdle to adopting TypeScript was getting everyone to learn TypeScript.

TypeScript works better the more types there are. If engineers aren't comfortable writing TypeScript code, fully adopting the language becomes an uphill battle.

Etsy has several hundred engineers, and very few of them had TypeScript experience before the migration.

The strategy Etsy used was to onboard teams to TypeScript gradually on a team by team basis.

This had several benefits

- Etsy could refine their tooling and educational materials over time. Etsy found a course from [ExecuteProgram](#) that was great for teaching the basics of TypeScript in an interactive and effective way. All members of a team would have to complete that course before they onboarded.

- No engineer could write TypeScript without their teammates being able to review their code. Individual engineers weren't allowed to write TypeScript code before the rest of their team was ready.
- Engineers had plenty of time to learn TypeScript and factor it into their roadmaps. Teams that were about to start new projects with flexible deadlines were the first to onboard TypeScript.

How Khan Academy rewrote their Backend

Khan Academy recently went through a massive rewrite, where they replaced their old Python 2 monolith with a services-oriented backend written in Go.

Kevin Dangoor and Marta Kosarchyn are senior engineers at Khan Academy and they wrote a [series](#) of blog posts about the technical choices, execution and results of the rewrite. We'll be summarizing the series below.

Summary

In late 2019, Khan Academy was looking to upgrade their backend. The site was built on a Python 2 monolith and it worked well for over 10 years.

However, Python 2 was about to reach the official [end of life on January 1rst, 2020](#) so Khan Academy engineers decided they had to update.

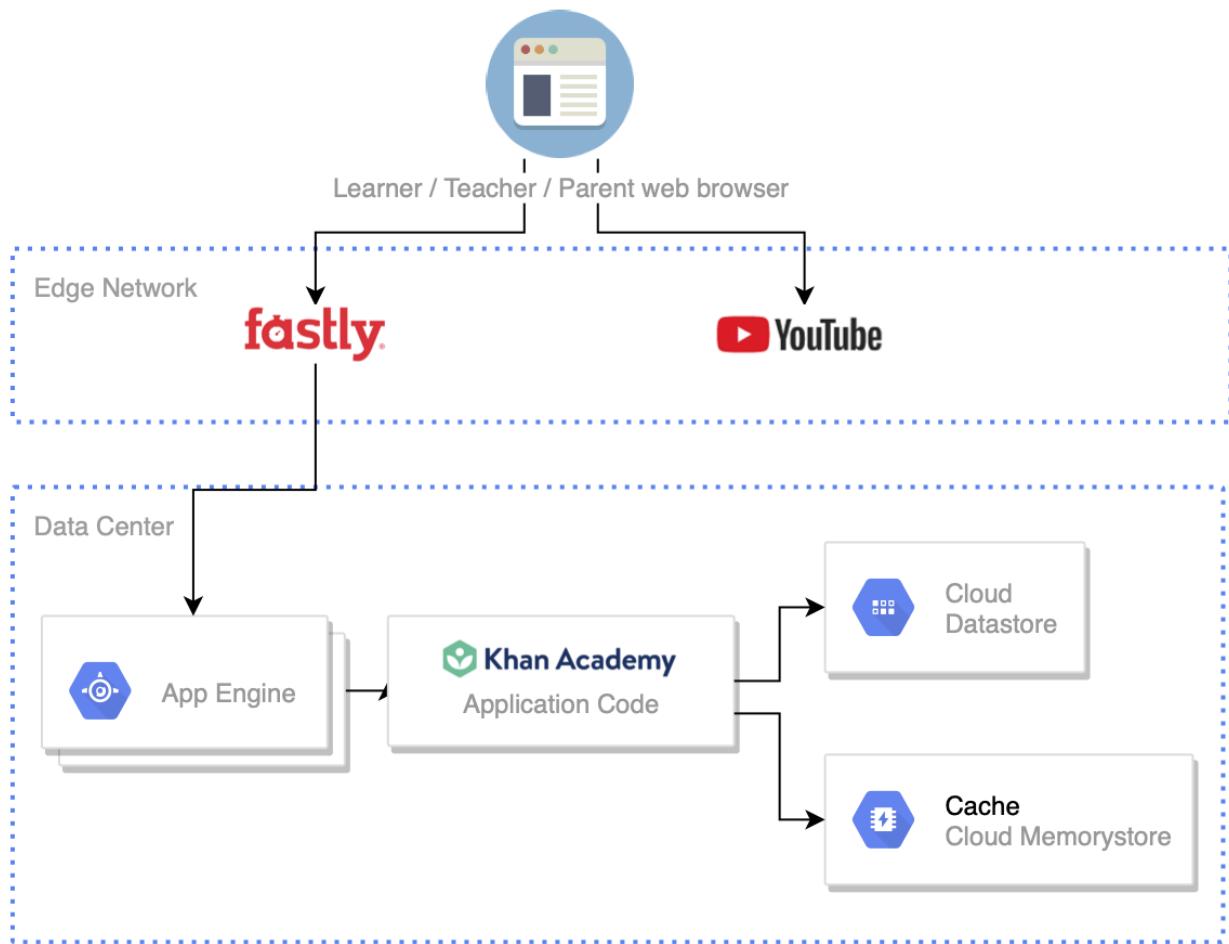
KA (Khan Academy) had several options

- Migrate from Python 2 to Python 3 - This would get KA a 10-15% boost in backend server code performance and Python 3's language features.
- Migrate from Python 2 to Kotlin - [KA started using Kotlin](#) for compute-intensive backend tasks since it was more performant than Python 2. Switching from Python to Kotlin could mean Khan Academy becomes more responsive and server costs go down.
- Migrate from Python 2 to Go - Go is a simple and concise language with very quick compilation times, first-class support on [Google App Engine](#) and less memory usage than Kotlin (based on KA's tests).

Of these options, Khan Academy decided to go with the third choice and do a rewrite of their Python 2 monolith with Go.

They ran performance tests and found that Go and Kotlin (on the JVM) perform similarly, with Kotlin being a few percent ahead. However, Go used a lot less memory.

The dramatic performance difference between Go and Python made the effort involved in the switch worth it.



Brief Overview of Go

Go is a statically typed, compiled programming language that is syntactically similar to C (it's often described as C for the 21rst century). Go was designed at Google by [Ken Thompson](#), [Rob Pike](#) and [Robert Griesemer](#).

Ken Thompson and Rob Pike were key employees at Bell Labs, and were instrumental in building the original Unix operating system (and a bunch of other stuff, they developed the UTF-8 encoding for example).

Go includes things like garbage collection, [structural typing](#), and [extremely fast compile times](#).

You can learn about the design of Go from this [article](#) by Rob Pike.

Monolith to Services

In addition to switching to Go, Khan Academy decided they would switch to a services-oriented architecture.

Previously, all of Khan Academy's servers ran the same code and could respond to a request for any part of the website. Separate services were used for storing data and managing caches, but the logic for any request was the same regardless of which server responded.

Despite the additional complexity that comes with a services architecture, Khan Academy decided to go with it because of several big benefits.

- Faster Deployments - Services can be deployed independently, so deployment and test runs can move more quickly. Engineers will be able to spend less of their time on deployment activities and get changes out more quickly when needed.
- Limited Impact for Problems - KA engineers can now be more confident that a problem with a deployment will have a limited impact on other parts of the site.
- Hardware and Configuration - By having separate services, engineers can now choose the right kinds of instances and hosting configuration needed for each service. This helps to optimize both performance and cost.

Despite the change in architecture, Khan Academy plans to continue using Google App Engine for hosting, Google Cloud Datastore for their database, and other Google Cloud products.

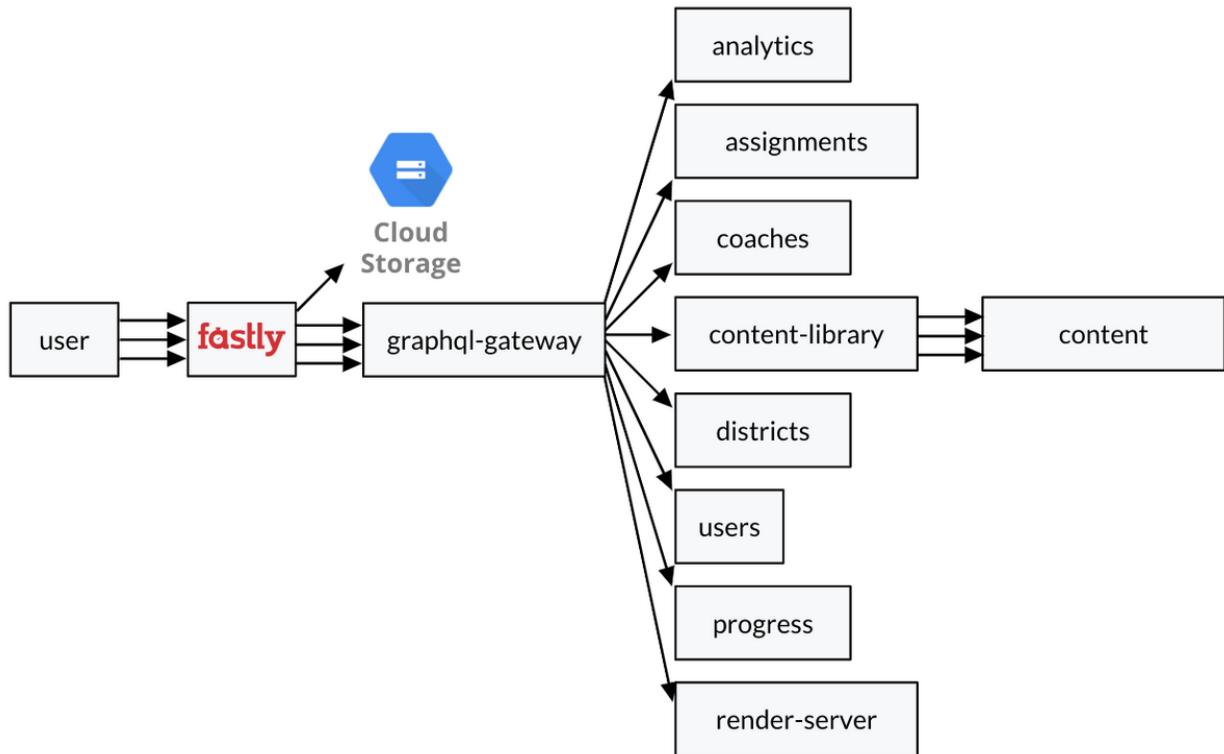
The Implementation

In the [next part](#) of the series, Kevin Dangoor talks about how Khan Academy rewrote their backend without downtime.

Big rewrites are extremely risky (Joel Spolsky, co-founder and former CEO of Stack Overflow, has a great blog post on [this](#)) so KA picked a strategy of *incremental rewrites*.

The hub of Khan Academy's new backend is based on [GraphQL Federation](#). Khan Academy is switching from using REST to GraphQL, and GraphQL Federation allows you to combine multiple backend services into one unified graph interface.

This way, you have a single, typed schema for all the data the various backend systems provide that is accessed through the GraphQL gateway. Each backend service provides part of the overall GraphQL schema and the gateway merges all of these separate schemas into one.



Khan Academy incrementally switched over from the old backend to the new backend.

In order to make sure that the new service is working correctly, they would query *both* the new distributed backend and the old monolith and compare the results and then return one of them.

KA had a 4 step process for handling the switch-over.

1. The monolith is in control - At first, the services backend will not have the functionality to answer a specific request, so the GraphQL gateway will route that request to the Python monolith. This request will be noted and KA engineers can write the Go code to handle the request in the new backend.
2. Side-by-side - Once the new backend can handle the request, KA engineers will switch to side-by-side. In this state, the GraphQL gateway will call *both* the Python code and the new Go code. It will compare the results, log the instances where there's a difference, and return the Python result to the user.
3. Migrated - After lots of testing, the request's status will be upgraded to *migrated*. At this point, the gateway will *only* send traffic to the Go service. In case something goes wrong, the Python code is still there and ready to respond.
4. Python code removed - Finally, after extensive testing, the Python code is removed.

Khan Academy used this process to rewrite their backend with high availability. They were still able to handle this task *despite* having a massive increase in website traffic.

The bulk of the rewrite was done in 2020, when schools switched to remote due to COVID-19 and students, parents and teachers made significantly more use of Khan Academy.

Within a period of 2 weeks, KA saw an increase of 2.5x in usage. You can read about how they handled it in this [blog post](#) by Marta Kosarchyn.

Final Results

Early this month, Marta Kosarchyn published a [blog post](#) detailing the final results of Khan Academy's rewrite.

As of August 30th, 2021, the new services backend was handling 95% of all traffic to the site.

They were able to meet their initial estimate of completion that they made 20 months prior (despite the massive bump with COVID) and achieve performance goals for the new backend.

They relied on several principles to make the process go smoothly.

- Avoid Scope Creep - At every turn, engineers sought to do as direct a port as possible from Python to Go, while still ending up with code that read like Go code. Khan Academy's codebase had areas that they wished were structured differently, but if engineers tried to tackle those issues at the same time as the Go rewrite, they would never finish.
- Side by side testing - We've already discussed this, but the side-by-side testing approach was critical to the success of the rewrite. It was an efficient way to make sure that the functionality being replaced was equivalent.
- Borderless Engineering - Engineers would work on various product code areas and new services, stepping beyond the borders of their usual product area responsibility. This had to be done carefully, making sure that engineers would spend sufficient time to learn a new service area to be able contribute thoughtfully. It meant that engineers would sometimes switch teams or service ownership would transfer between teams.

Redesigning Etsy's Machine Learning Platform

Kyle Gallatin and Rob Miles are two software engineers at Etsy working on the Machine Learning Platform team.

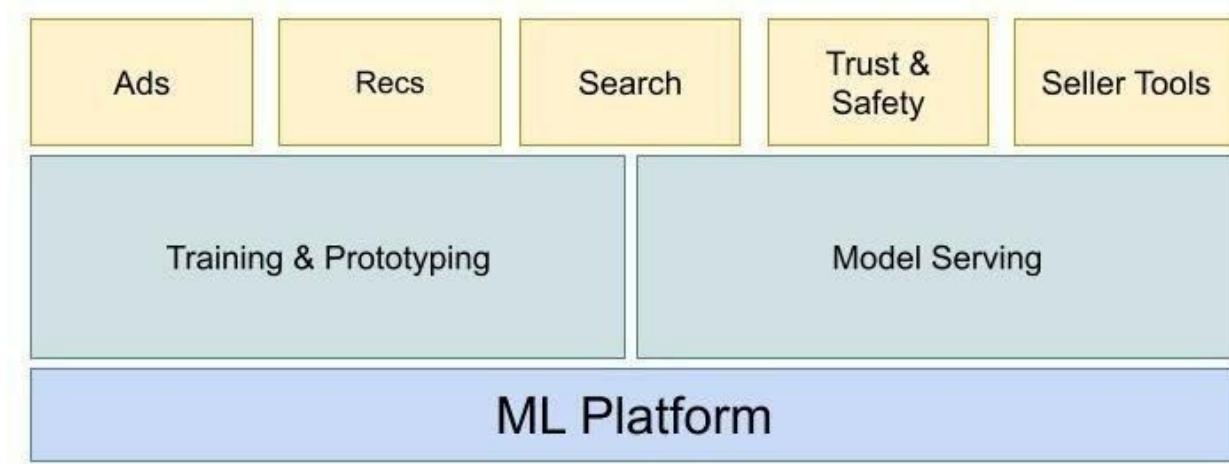
They wrote a great [article](#) summarizing the design of the ML infrastructure that Etsy uses and the design choices that went into building the system.

Here's a summary

Etsy is an e-commerce platform that allows users to sell handmade or vintage items. Popular products sold on the site include things like jewelry, clothing, bags, etc.

The website makes extensive use of machine learning models for things like search, recommendations, the ad platform, trust & safety, and more.

The ML Platform team at Etsy develops and maintains the technical infrastructure that Etsy data scientists use to prototype, train and deploy ML models at scale.



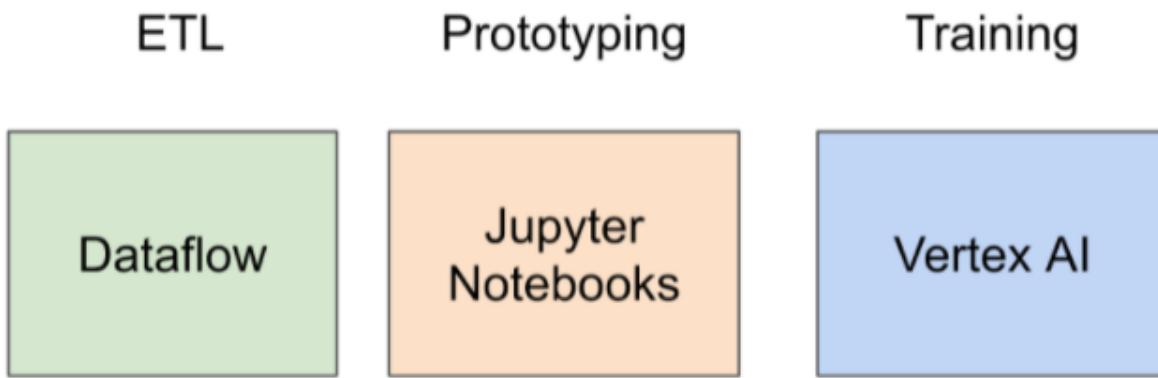
Etsy's first ML platform was built in 2017, when the data science team was much smaller and largely relied on much simpler models. As the platform had to start supporting more complex machine learning projects and new ML frameworks, the maintenance costs started to become too high.

They decided they would build a new version of the ML platform, and would rely on the following principles.

- Avoid in-house tooling - Use open-source technologies like TensorFlow and managed ML solutions from platforms like Google Cloud. This way, the data science team can build models quickly without having to rely on the platform team for support.
- Embrace self-service - Instead of burdening the data science team with platform-specific abstractions, let the open source and managed tools and technologies speak for themselves. Users of the ML platform can rely on those tool's well-written documentation and free up the ML Platform team to focus away from support and more on core work.
- Toolset Flexibility - TensorFlow is the primary modeling framework, however users of the platform shouldn't be limited to a single toolset. They should be able to experiment and deploy models using any ML library.

The design of ML Platform V2

Training and Prototyping

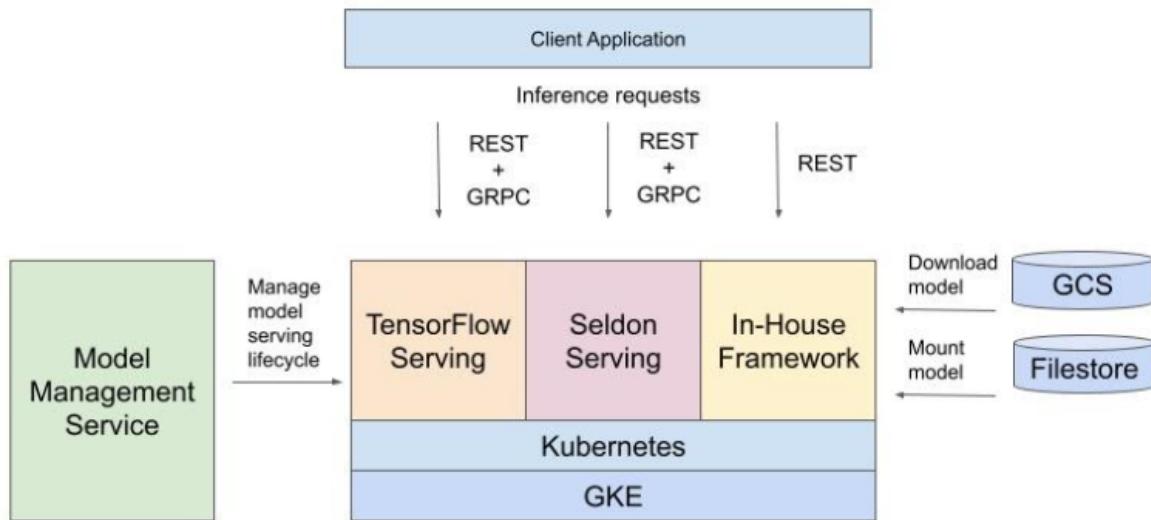


Etsy's training and prototyping platform largely relies on Google Cloud services like [Vertex AI](#) and [Dataflow](#), where the data science team can experiment freely with the ML framework of their choice.

They can use Jupyter Notebooks to quickly iterate while using complex infrastructure and managing large amounts of data.

Massive extract transform load (ETL) jobs can be run through Dataflow while complex training jobs can be submitted to Vertex AI for optimization.

Model Serving



Etsy relies on [Google Kubernetes Engine \(GKE\)](#) for the core of their Model Serving system (making inferences in production).

To deploy models, data scientists will create stateless ML microservices that are deployed in Etsy's Kubernetes cluster.

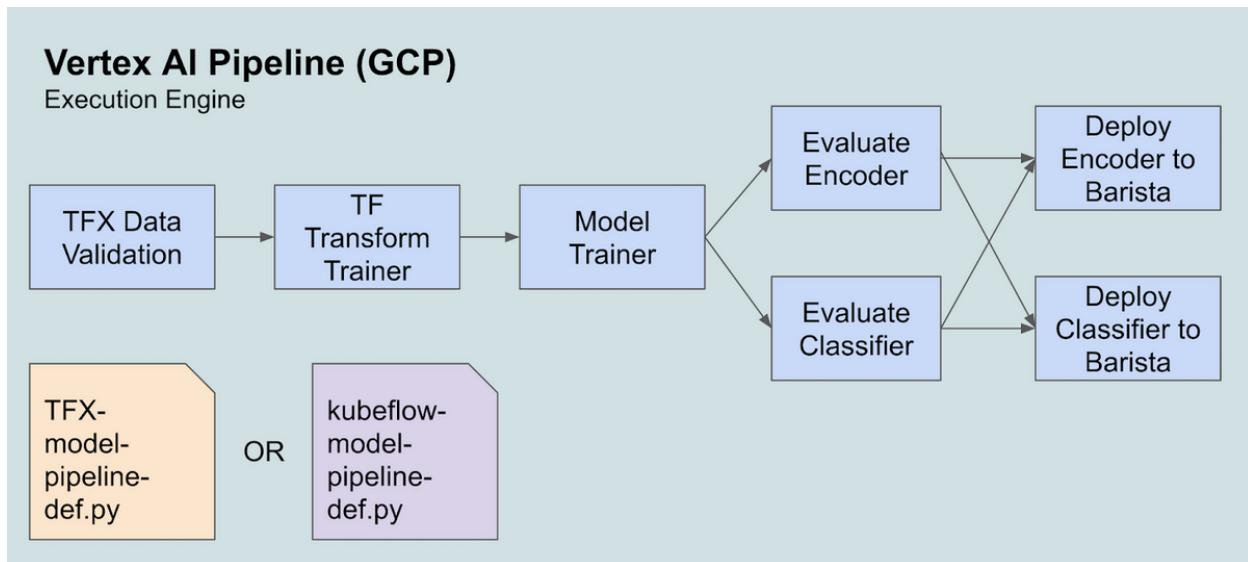
These microservices will then serve requests from Etsy's website or mobile app.

The deployments are managed through the Model Management Service, an in-house developed control plane that gives the data science team a simple UI to manage their model deployments.

The Model Management Service violates Etsy's *avoid in-house* rule, but that's because it was already built-out and the ML platform team found it was still the best tool available.

However, they extended the Model Management Service to support two additional open-source serving frameworks: [TensorFlow serving](#) and [Seldon Core](#).

Workflow Orchestration



In order to keep ML models up-to-date, the ML platform also needs robust pipelines for retraining and deployment.

Etsy relies on [Kubeflow](#) and [TFX pipelines](#) (TensorFlow Extended) for this. With Google Cloud Platform's Vertex AI Pipelines, the data science team can develop and test pipelines using either the Kubeflow or TFX SDE, based on their own preference.

This makes it much faster to write, test and validate pipelines.

Outcomes

The ML Platform team estimates that with V2, ML practitioners at Etsy can now go from idea to live ML experiment in half the time it previously took. Launching new model architectures takes days instead of weeks, and data scientists can launch dozens of hyperparameter tuning experiments with a single command.

The biggest challenge around V2 has been encouraging adoption of the new ML platform. Migrating to a new platform requires upfront effort that may not align with the current priorities of the staff at Etsy.

In order to encourage adoption, the ML Platform team provided additional support to early adopters to ease the transition, put a big emphasis on transparency around components and new features, and showcased best practices.

For more details, read the full [blog post](#)!

How Video Works

[Leandro Moreira](#) is a Lead Software Engineer at Globo where he works on their live video streaming platform and infrastructure.

He wrote a [great blog post](#) on all the engineering behind how videos play on your computer (adaptive bitrate streaming, HLS, etc.), how videos are delivered to your computer (CDNs, Multi-CDNs, etc.) and how film is processed into digital video (Codecs, Containers, FFmpeg, etc.).

Here's a summary of some parts from the post

Playback

When you come across a website that has a video player embedded in it, there's quite a bit going on behind the scenes.

You have the player UI, with the pause/play button, subtitle controls, video speed and other options.

Players will support different options around DRM, ad injection, thumbnail previews, etc.

Behind the scenes, modern video platforms will use adaptive bitrate streaming to stream the video from the server.

Adaptive bitrate streaming means that the server has several different versions of the video (known as renditions) and each version differs in display size (resolution) and file size (bitrate).

The video player will dynamically choose the best rendition based on the user's screen size and bandwidth. It will choose the rendition that minimizes buffering and gives the best user experience.

HLS

[HTTP Live Streaming \(HLS\)](#) is a protocol designed by Apple for HTTP-based adaptive bitrate streaming. It's the most popular streaming format on the internet.

The basic concept is that you take your video file and break it up into small segments, where each segment is 2-12 seconds long.

If you have a 2 hour long video, you could break it up into segments that are 10 seconds long and end up with 720 segments.

Each of the segments is a file that ends with a .ts extension. The files are numbered sequentially, so you get a directory that looks like this

segments/

00001.ts

00002.ts

00003.ts

00004.ts

00005.ts

The player will then download and play each segment as the user is streaming. It will also keep a buffer of segments in case the user loses network connection.

Again, HLS is an *adaptive bitrate streaming* protocol, so the web server will have several different renditions (versions) of the video that is being played.

All of the renditions will be broken into segments of the same length. So, going back to our example with the 2 hour long video, it could have 720 segment files at 1080p, 720 segment files at 720p, 720 segment files at 480p.

All the segment files are ordered and are each 10 seconds in length.

The player will then look at the amount of bandwidth available and make the best guess as to which rendition's segment file it should download next.

If your network connection slows down while you're watching a video, the player can downgrade you to a lower quality rendition for the next segment files.

When your connection gets faster, the player can upgrade your rendition.

MP4 & WebM

An alternative is taking an HTML

This is called *pseudo-streaming* or *progressive download*, where the video file is downloaded to a physical drive on the user's device.

Typically, the video is stored in the temporary directory of the web browser and the user can start watching while the file is being downloaded in the background.

The user can also jump to specific points in the video and the player will use byte-range requests to estimate which part of the file corresponds to the place in the video that the user is attempting to seek.

What makes MP4 and WebM playback inefficient is that they do not support adaptive bitrates.

Every user who wants to watch the file buffer-free must have an internet connection that is fast enough to download the file faster than the playback.

Therefore, when you are using these formats you have to make a tradeoff between serving a higher quality video file vs. decreasing the internet connection speed requirements.

Delivery

When delivering video to your user, there's two primary components

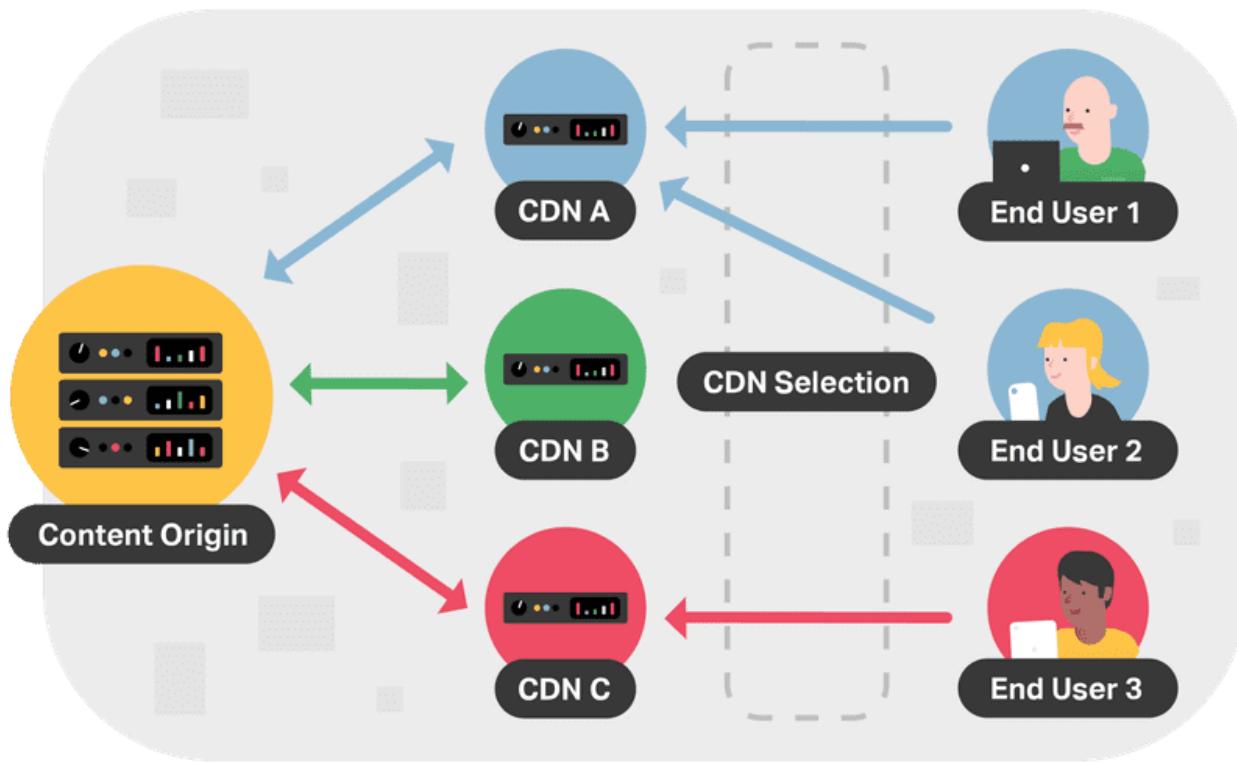
- the origin server
- the content delivery network

The origin server is the source of truth. It's where the developer uploads the original video files.

The CDN will then pull files from the origin server and cache that file on a bunch of interconnected servers around the world (in locations that are close to your users).

That way, when users want to request the file, they can do so from a server in the CDN. This is way faster (and much more scalable) than your origin server sending the entire file to all your users.

Many enterprises will choose a Multi-CDN environment, where the load is distributed among multiple CDNs. This improves the user experience by giving them more servers to choose from and improving the availability of your website.



This is a brief summary from the blog post.

You can read the full post [here](#).

How Grab Processes Billions of Events in Real Time

Grab is the largest transportation and food delivery company in Southeast Asia with more than 25 million monthly users completing ~2 billion transactions per year.

One of the marketing features in the Grab app is to offer real-time rewards whenever a user takes a certain action (or series of actions).

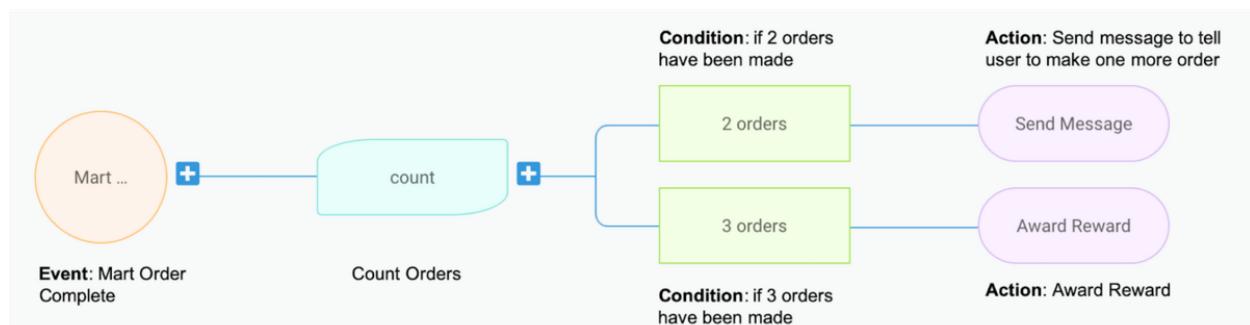
For example, if a user uses the Grab app to get a ride to work in the morning, the app might immediately reward her with a 50% off ride reward that she can use in the evening for the ride back home.

Jie Zhang and Abdullah Al Mamum are two senior software engineers at Grab and they wrote a great [blog post](#) on how they process thousands of events every second to send out hundreds of millions of rewards monthly.

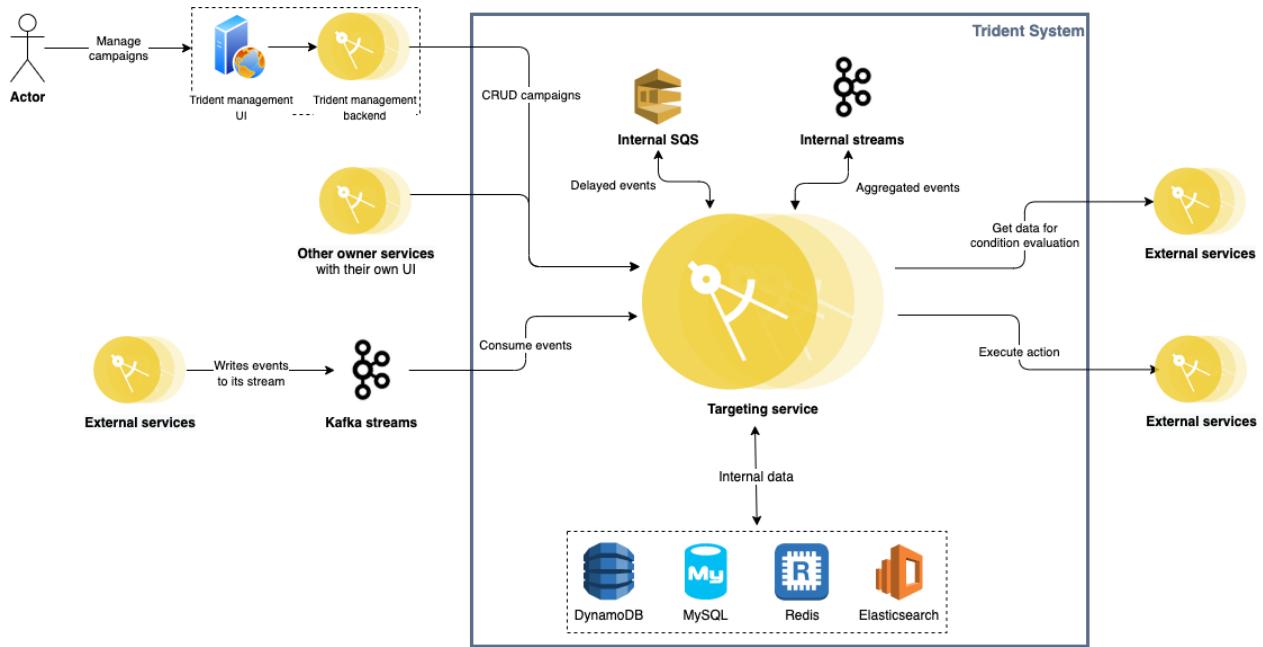
Here's a summary

Grab runs growth campaigns where they'll reward a user with discounts and perks if the user completes a certain set of actions. Over a typical month, they'll send out ~500 million rewards and over 2.5 billion messages to their end-users.

Trident is the engine Grab engineers built to handle this workload. It's an *If This, Then That* engine which allows Grab's growth managers to create new promotional campaigns. *If a user does this, then award that user with that.*



The Architecture of Trident



Trident's architecture was designed with the following goals

- Independence - Trident must run independently of other backend services, and it should not bring performance impacts to downstream backend services.
- Robustness - All events must be processed *exactly* once. No events can be missed and events should not be processed multiple times.
- Scalability - Trident must be able to scale up processing power when volume on the Grab platform surges.

Whenever a customer uses one of Grab's products the backend service associated with that product will publish an event to a specific Kafka stream.

Trident subscribes to all the events from these multiple Kafka streams and processes them. By utilizing Kafka streams, Trident is decoupled from the upstream backend services.

Kafka guarantees *at-least-once* message delivery and then Trident makes sure any duplicate events are filtered out. This gives Trident *exactly-once* event processing, fulfilling the robustness criteria.

After filtering out duplicates, Trident will process each event and check if it results in any messages/rewards that have to be sent to the user. Trident does this by taking the event and running a rule evaluation process where it checks if the event satisfies any of the pre-defined rules set by the growth campaigns.

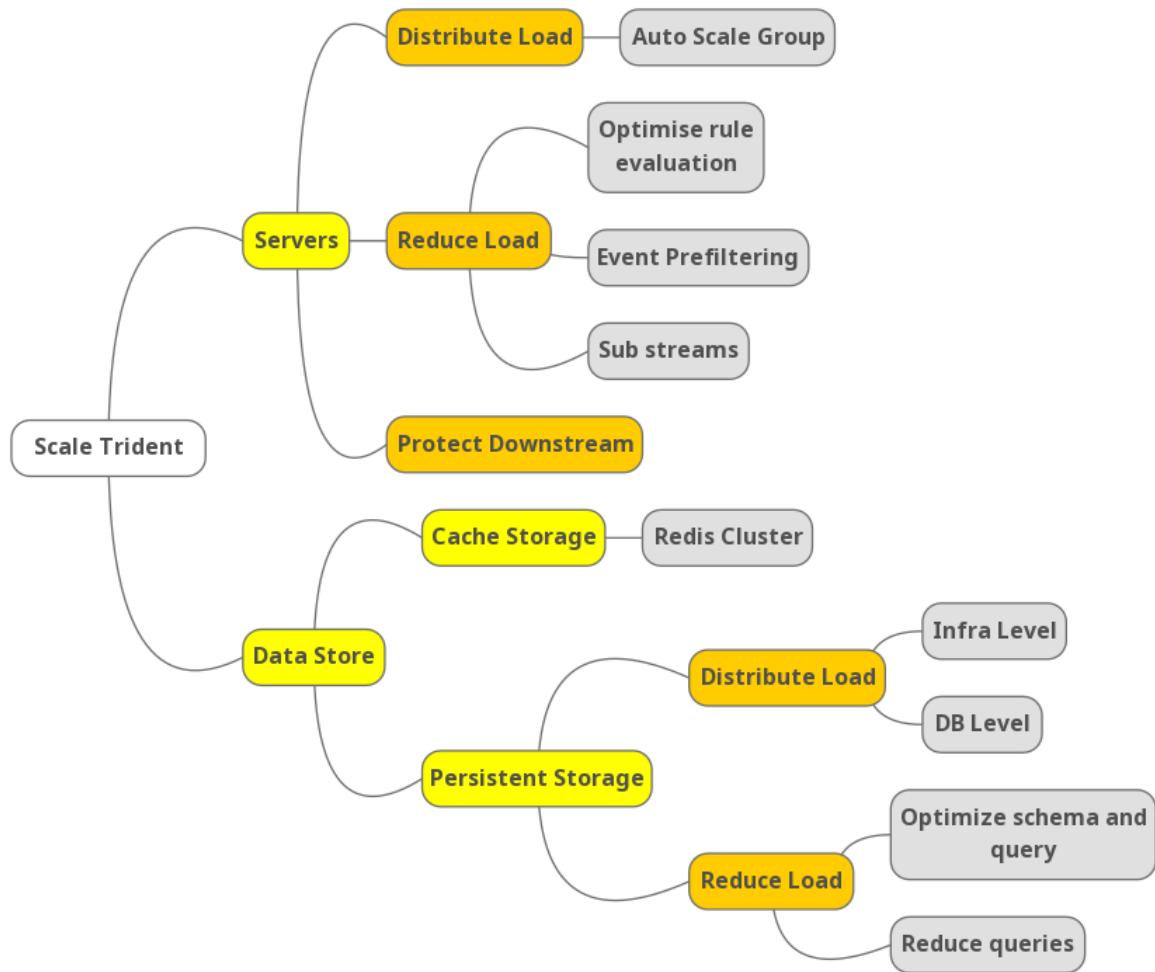
All processed events are stored in Redis (for 24 hours) and events that trigger an action are persisted in MySQL as well.

If an action is triggered, Trident will then call the backend service associated with that action. These calls are rate-limited (with tighter limits during peak hours) so that Trident doesn't accidentally DoS attack any of Grab's downstream backend services.

Scalability

The number of events that Trident has to process can vary widely based on the time of day, day of week and time of year. During the peak of 2020, Trident was processing more than 2,000 events per second.

Grab uses quite a few strategies to make sure Trident can scale properly. The strategies are illustrated in this diagram.



It boils down to two things: scaling the server level and scaling the data store level.

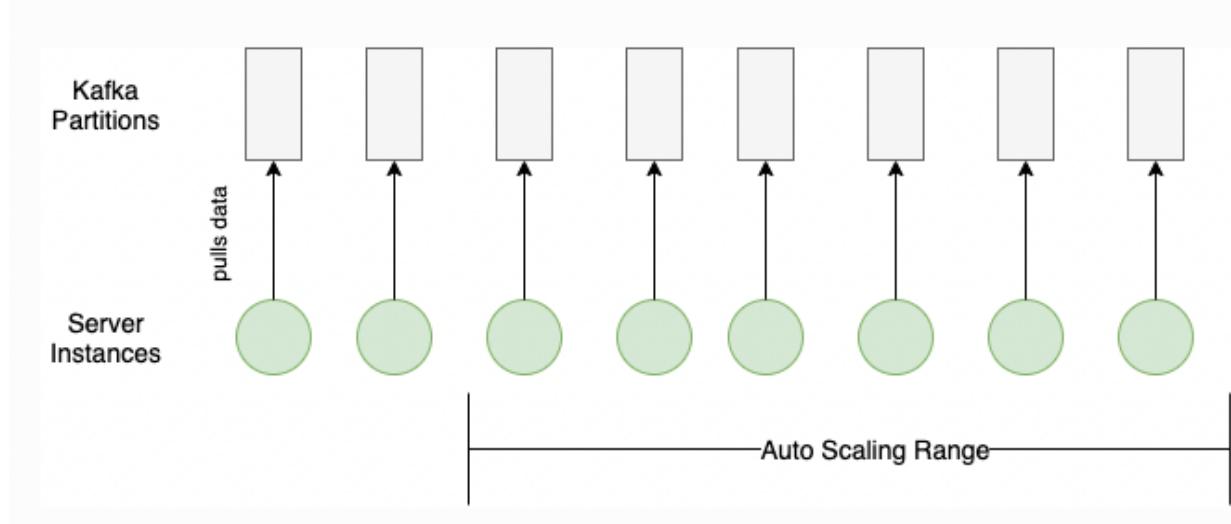
Scaling the Server Level

The source of events for Trident are Kafka streams. Upstream backend services that are handling delivery/taxi orders will publish events to these streams after they handle a user's request.

Trident can handle increased load (more events coming down the Kafka streams) by

- Auto-scaling horizontally - Grab can add more server instances to handle Trident's workload. However, they have to be careful and make sure that load

is being distributed evenly across the server instances by matching kafka partitions with the server auto-scaling.



- Reducing Load - The majority of the processing that the Trident servers are doing is checking to see if the event matches the criteria for any of the campaigns and whether any actions are triggered. Grab engineers sped this process up by prefiltering events. They load active campaigns every few minutes and organize them into an in-memory hashmap with the event type as the key and the list of corresponding campaigns as the value. When processing an event, they can quickly figure out all the possible matching campaigns by first checking in the hash map.

If any actions are triggered, Trident will call downstream backend services to handle them. For example, the GrabRewards service could be called to give a user a free ride.

There are strict rate-limits built in to stop Trident from overwhelming these downstream services during a time of high load.

Scaling the Data Store Level

Trident uses two types of storage: cache storage (Redis) and persistent storage (MySQL and S3).

Scaling cache storage isn't too complicated since Redis Cluster makes it pretty simple. Engineers can add new shards at any time to spread out the load and data replication prevents any data loss from shard failures.

In terms of persistent storage, Trident has two types of data in terms of access pattern: online data and offline data.

The online data is frequently accessed (so it has to be relatively quick) and medium size (a couple of terabytes). Grab uses MySQL for this data.

The offline data is infrequently accessed but very large in size (hundreds of terabytes generated per day). Grab uses AWS S3 for this.

For the MySQL database, Grab added read replicas that can handle some of the load from the read queries. This relieved more than 30% of the load from the master instance and allows MySQL queries to be performant.

In the future, they plan to vertically partition (split the database up by tables) the single MySQL database into multiple databases based on table usage.

[For more details, read the full blog post here.](#)

Lessons Learned from Implementing Payments in the DoorDash Android App

Harsh Alkutkar is a software engineer on DoorDash's ordering experience team. He wrote a great [blog post](#) on *Eight Things We Learned from Implementing Payments in the DoorDash Android App*

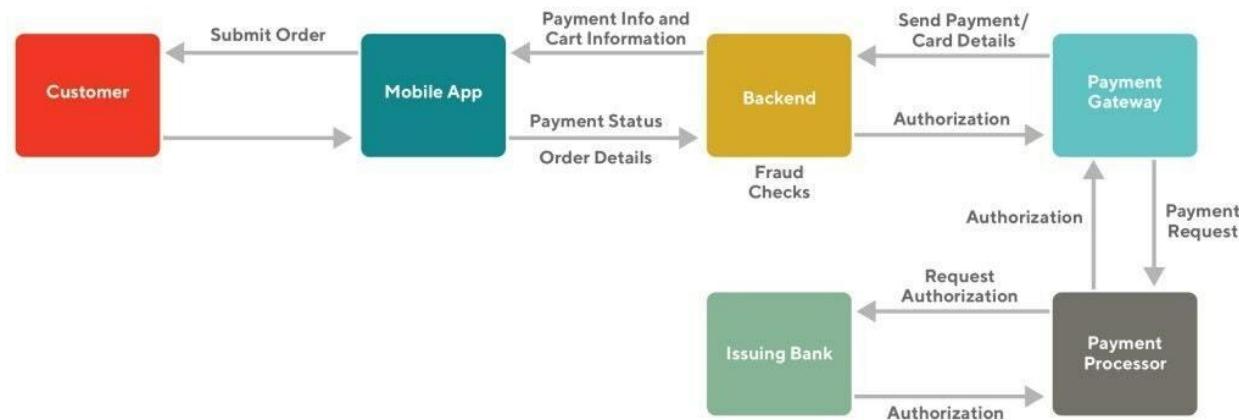
Summary

How mobile payments are typically implemented

When a user is making an online order, they will submit their credit card information to a payment gateway such as Stripe or PayPal. The gateway encrypts this information and facilitates the transaction with payment processors.

The payment processor will talk to the issuing bank (for the user's credit card) and request approval.

The approval will then bubble to the backend, which lets the client know if the payment was accepted/declined.



Mobile payments can become very complex for the following reasons

- Multiple Payment Methods - In order to boost conversion rates, you want to offer as many payment methods as possible to the user. Each method requires its own integration into the app *and* requires its own custom testing strategy.
- User experience - The UI needs to work with *all* payment methods and also for new and existing users. This creates quite a few scenarios that have to be implemented and tested.
- Testing - Testing cannot be an afterthought. The backend has to be designed in a way that allows every payment method and flow to be tested before a release.
- Fraud - Anti-fraud measures need to be implemented in any app that includes a mobile payment component.
- Location - you need to account for the user's location before processing their payment to comply with each country's laws and regulations

Here's a couple of the lessons DoorDash engineers learned while implementing payments in the DoorDash Android app.

Plan and design for future payment methods

In an earlier version of the DoorDash app, the developers didn't properly account for the addition of new payment methods to the app. Instead, it was designed around credit cards and Google Pay. This led to challenges when adding new methods like PayPal.

In the new design, engineers introduced the notion of payment methods into the codebase. These are categorized into local payment methods that are part of the device (like Google Pay) and external payment methods that require interactions with a payment gateway (like Stripe).

Beware of restrictions and implementation guidelines specific to payment methods

Payment vendors may have specific ways they want to be portrayed in an app.

For example, Google Pay requires that it be the [primary payment option wherever possible](#).

Additionally, they have strict UX guidelines that explain how to display their logos and buttons.

Plan for consumers in different countries or traveling consumers

Payments usually can't be implemented in a generic way that scales worldwide. Each country has its own technical, legal and accounting implications.

Some payment methods may also need extra verification or information in other countries.

Plan for performance

It's absolutely critical to keep an app performant while it processes payments. Caching the payment methods and cards makes it faster because there's less waiting for payment information on the cart and checkout screens.

However, this has to be done with care and must account for error cases where the backend and device are out of sync.

Add lots of telemetry

Payment flows can be tricky to debug if they don't work properly. Therefore, instead of just sending generic information like "*payment failed*", the system should send as much information as possible; including things like error codes from providers, device information or any diagnostics that can help to identify the state of the app when it failed.

However, be careful not to include any personal identifiable information or any payment information that could be compromised by an attacker.

[For more details on each of these points, read the full blog post here](#)

Why DoorDash migrated from Python to Kotlin

DoorDash is the largest food delivery app in the United States with more than 450 thousand restaurants, 20 million customers and 1 million deliverers.

Matt Anger is a Senior Staff Engineer at DoorDash where he works on the Core Platform and Performance teams.

He published a great [blog post](#) (May 2021) on DoorDash's migration from Python 2 to Kotlin. Here's a summary.

Summary

DoorDash was quickly approaching the limits of what their Django-based monolithic codebase could support.

With their legacy system, the number of nodes that needed to be updated added significant time to releases. Debugging bad deploys with [bisection](#) got harder and longer due to the number of commits each deploy had. The monolith was built with Python 2 which was also rapidly entering end-of-life.

Engineers at DoorDash decided to transition from the monolith to a microservices architecture. They also looked for a new tech stack to replace Python 2 and Django.

One of their goals was to only use one language for the backend.

Having one language would let them

- Promote Best Practices - Having one language makes it easier for teams to share development best practices across the entire company.
- Build Common Libraries - All engineers can share common libraries and tooling.
- Change Teams - Engineers can change teams with minimal friction, which encourages more collaboration.

Picking the Right Coding Language

First, DoorDash engineers looked at the parts of their tech stack that would not change.

They had a lot of experience with Postgres and Apache Cassandra, so they would continue to use those technologies as data stores.

They would use gRPC for synchronous service-to-service communication, with Apache Kafka as a message queue.

In terms of the programming language, the choices in contention were Kotlin, Java, Go, Rust and Python 3.

Here's the comparison they did...

Language	Pros	Cons
Kotlin	<ul style="list-style-type: none"> - Provides a strong library ecosystem - Provides first class support for gRPC, HTTP, Kafka, Cassandra, and SQL - Inherits the Java ecosystem. - Is fast and scalable - Has native primitives for concurrency - Eases the verbosity of Java and removes the need for complex Builder/Factory patterns - Java agents provide powerful automatic introspection of components with little code, automatically defining and exporting metrics and traces to monitoring solutions 	<ul style="list-style-type: none"> - Is not commonly used on the server side, meaning there are fewer samples and examples for our developers to use - Concurrency isn't as trivial as Go, which integrates the core ideas of gothreads at the base layer of the language and its standard library
Java	<ul style="list-style-type: none"> - Provides a strong library ecosystem - Provides first class support for GRPC, HTTP, Kafka, Cassandra, and SQL - Is fast and scalable - Java agents provide powerful automatic introspection of components with little code, automatically defining and exporting metrics and traces to monitoring solutions 	<ul style="list-style-type: none"> - Concurrency is harder than Kotlin or Go (callback hell) - Can be extremely verbose, making it harder to write clean code
Go	<ul style="list-style-type: none"> - Provides a strong library ecosystem - Provides first class support for GRPC, HTTP, Kafka, Cassandra, and SQL - Is a fast and scalable option - Has native primitives for concurrency, which make writing concurrent code simpler - Lots of server side examples and documentation is available 	<ul style="list-style-type: none"> - Configuring the data model can be hard for people unfamiliar with the language - No generics (but finally coming!) means certain classes of libraries are much harder to build in Go
Rust	<ul style="list-style-type: none"> - Very fast to run - Has no garbage collection but still memory and concurrency-safe - Lots of investment and exciting developments as large companies begin adopting the language - Powerful type system that can express complex ideas and patterns more easily than other languages 	<ul style="list-style-type: none"> - Relatively new, which means fewer samples, libraries, or developers with experience building patterns and debugging - Ecosystem not as strong as others - async/await was not standardized at the time - Memory model takes time to learn
Python 3	<ul style="list-style-type: none"> - Provides a strong library ecosystem - Easy to use - There was already a lot of experience on the team - Often easy to hire for - Has first class support for GRPC, HTTP, Cassandra, and SQL - Has a REPL for easy testing and debugging of live apps 	<ul style="list-style-type: none"> - Runs slowly compared to most options The global interpreter lock makes it difficult to fully utilize our multicore machines effectively - Does not have a strong type checking feature - Kafka support can be spotty at times and there are lags in features

After doing the comparison, they went with Kotlin. They had already done some testing around the language and it worked well.

Kotlin mitigated some of the pain points around Java like [Null Safety](#) and [Coroutines](#).

Some of the growing pains they faced with Kotlin were

- Educating DoorDash engineers on the language - Much of the online community around Kotlin is specific to Android dev, and there isn't as much content on backend engineering. To help engineers learn the language, they regularly held Lunch and Learn sessions and set up a slack channel for questions.
- Avoiding coroutine gotchas - DoorDash used gRPC for service-to-service communication however gRPC Kotlin wasn't available when they first made the switch. They used gRPC-Java, which lacked support for coroutines. gRPC Kotlin is now generally available so they made the migration to that. There are several other gotchas around coroutines that are discussed in the article.
- Getting around Java interoperability pain points - There were some pain points with Java interop. Many libraries claiming to implement modern [Java Non-blocking I/O standards](#) did so in an unscalable manner. This caused issues when using coroutines. Check the article for full details.
- Making dependency management easier - The build system and dependency management are a lot less intuitive than more recent solutions like Rust's Cargo or Go's modules. Some dependencies are particularly sensitive to version upgrades and can lead to issues where compilation succeeds but the app fails on boot up with odd, seemingly irrelevant back traces. DoorDash engineers learned which projects tend to cause these issues most often and have guidelines for how to catch and bypass them.

[*For more details, read the full article*](#)

Clock Synchronization and NTP

When you're talking about measuring time, the gold standard is an atomic clock. Atomic clocks have an error of ~1 second in a span of 100 million years.

However, atomic clocks are way too expensive and bulky to put in every computer. Instead, individual computers contain quartz clocks which are *far less* accurate.

The [clock drift](#) differs on the hardware, but it's an error of ~10 seconds per month.

When you're dealing with a distributed system with multiple machines, it's very important that you have *some degree* of clock synchronization. Having machines that are dozens of seconds apart on time makes it impossible to coordinate.

Clock skew is a measure that tells you the difference between two clocks on different machines at a certain point of time.

You can never reduce clock skew to 0, but you want to reduce clock skew as much as possible through the synchronization process.

The way clock synchronization is done is with a protocol called [NTP, Network Time Protocol](#).

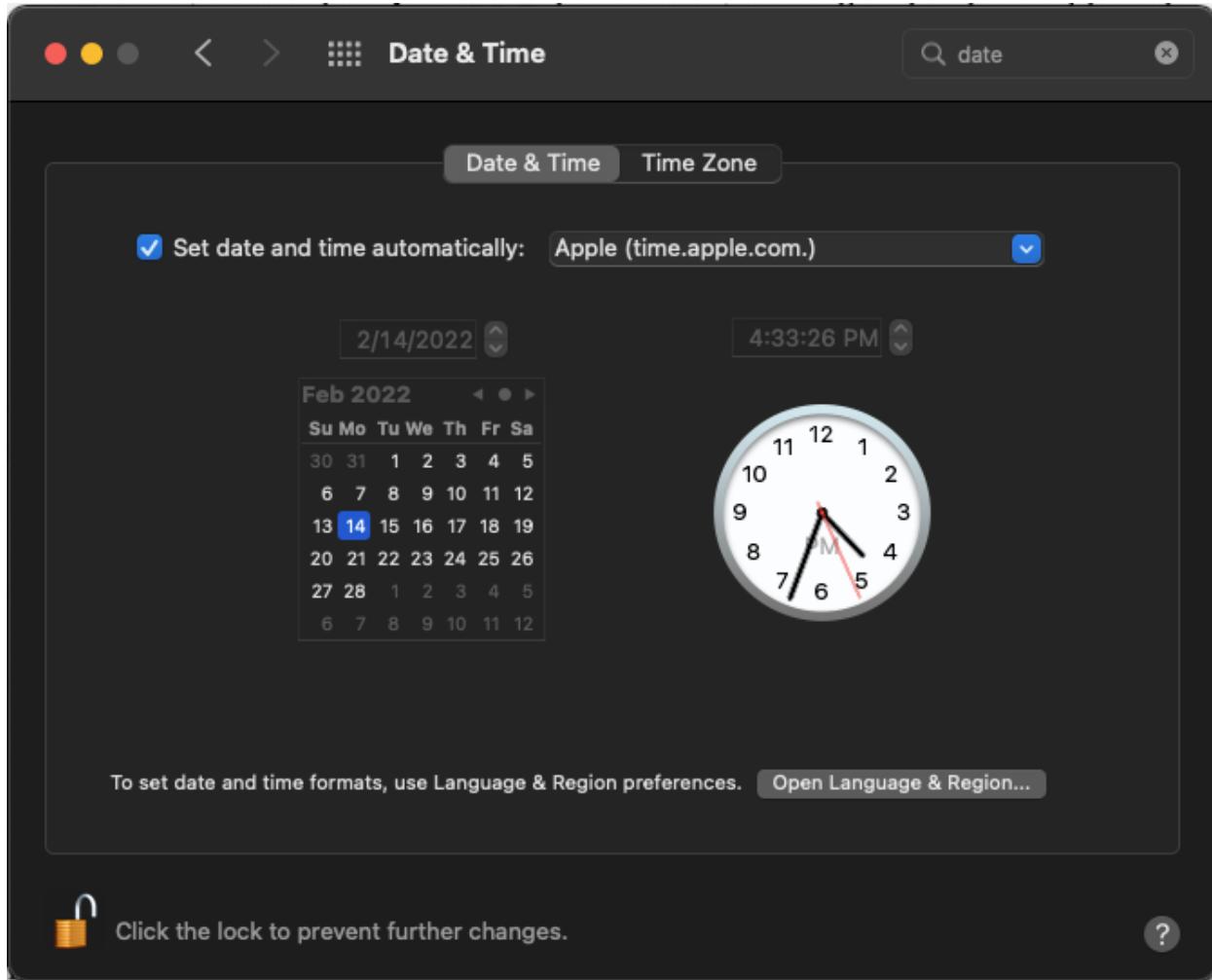
NTP works by having servers that maintain accurate measures of the time. Clients can query those servers and ask for the current time.

The client will take those answers, discard any outliers, and average the rest. It'll use a variety of statistical techniques to get the most accurate time possible.

This is a great [blog post](#) that delves into the clock synchronization algorithm.

[Here's a list](#) of NTP servers that you can query for the current time. It's likely that your personal computer uses NTP to contact a time server and adjust its own personal clock.

My personal computer uses *time.apple.com* as its NTP time server.



It's not possible for every computer in the world to directly query an atomic clock since there aren't enough atomic clocks to satisfy that demand.

Therefore, there are some NTP servers in between your computer and the reference clock.

NTP arranges these servers into strata

- Stratum 0 - atomic clock
- Stratum 1 - synced directly with a stratum 0 device
- Stratum 2 - servers that sync with stratum 1 devices
- Stratum 3 - servers that sync with stratum 2 devices

And so on until stratum 15. Stratum 16 is used to indicate that a device is unsynchronized.

A computer may query multiple NTP servers, discard any outliers (in case of faults with the servers) and then average the rest.

Computers may also query the same NTP server multiple times over the course of a few minutes and then use statistics to reduce random error due to variations in network latency.

For connections through the public internet, NTP can usually maintain time to within tens of milliseconds (a millisecond is one thousandth of a second).

To learn more about NTP, watch the [full lecture by Martin Kleppmann](#).

Building Faster Indexing with Apache Kafka and Elasticsearch

DoorDash is the largest food delivery app in the United States with more than 20 million consumers and 450 thousand restaurants.

A critical part of the DoorDash app is the search function. You can search for Scallion Pancakes and the DoorDash app will give you restaurants near you that are open and currently serving that dish.

Solving this problem at scale is quite challenging, as restaurants are constantly changing their menus, store hours, locations, etc.

You need to quickly index all of the store data to provide a great restaurant discovery feature.

Satish, Danial, and Siddharth are software engineers on DoorDash's Search Platform team, and they wrote a great [blog post](#) about how they built a faster indexing system with Apache Kafka, Apache Flink and Elasticsearch.

Here's a summary

DoorDash's Problem with Search Indexing

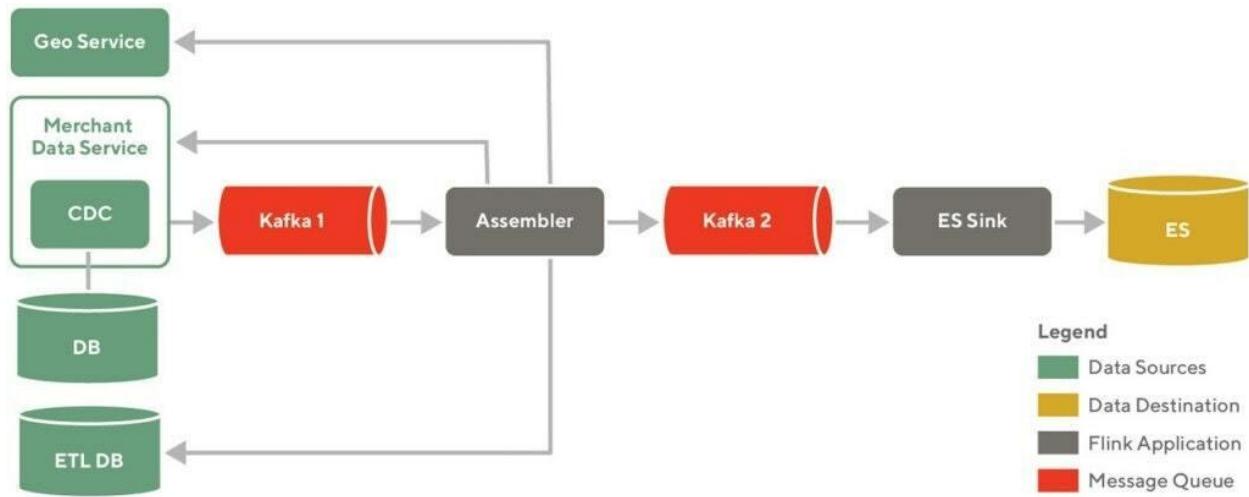
DoorDash's legacy indexing system was very slow, unreliable and not extensible. It took a long time for changes in store and item descriptions to be reflected in the search index. It was also very difficult to assess the indexing quality.

There were frequent complaints about mismatches in store details between the search index and the source of truth. These had to be fixed manually.

The New System

Engineers solved these problems by building a new search indexing platform with the goals of providing fast and reliable indexing while also improving search performance.

The new platform is built on a data pipeline that uses Apache Kafka as a message queue, Apache Flink for data transformation and Elasticsearch as the search engine.



The components of the architecture are

- Data sources - These are the sources of truth for the data. When [CRUD](#) operations take place on the data (changing store menu, updating store hours, etc.) then they are reflected here. DoorDash uses [Postgres](#) as the database and [Snowflake](#) as the data warehouse.
- Data destination - DoorDash is using [Elasticsearch](#) here as the final data destination. It will serve as the data store and search engine.
- Flink application - There are two custom Apache Flink applications in this pipeline: Assembler and ES Sink. Assembler is responsible for assembling all the data required in an Elasticsearch document. ES Sink is responsible for shaping the documents as per the schema and writing the data to the targeted Elasticsearch cluster.
- Message queue - Kafka 1 and Kafka 2 are the message queue components.

This data pipeline allows for fast, incremental changes to the search index when there are changes to the restaurant data.

The changes in data sources are propagated to Flink applications using Kafka. The Flink apps implement business logic to curate the search documents and then write them to Elasticsearch.

Incremental Indexing

The indexing pipeline processes two main types of data changes.

The first type of data change is when human operators make ad hoc changes to stores or restaurant items. An example of a possible data change is a restaurant owner adding a new dish to her menu.

The second type of data change is [ETL](#) data changes that are generated from machine learning models. Things like restaurant ratings/scores or auto-generated tags are generated by machine learning models and then stored in a data warehouse.

Both of these changes need to be reflected in the search index for the best customer experience.

Here's how DoorDash does it.

Indexing Human Operator Changes

Restaurant owners will frequently update their menus and store information. These changes need to be reflected onto the search experience as quickly as possible.

The updates are saved in data stores like Postgres.

To keep track of these updates, DoorDash search engineers rely on [Change Data Capture \(CDC\)](#) events.

DoorDash engineers implemented save hooks in the application to propagate change events to Kafka whenever there is a change on the underlying data store.

After receiving the Kafka events, the Assembler app will make backend calls to gather more information about the change and to create an event which it pushes to Kafka for the ES Sink app to consume.

They tested other solutions like [Debezium connector](#), a Red Hat-developed open source project for capturing row-level changes with Postgres but they found that this strategy had too much overhead and was not performant.

Indexing ETL data

Many properties that are used in the search index are generated by ML models. Things like restaurant scores, auto-generated tags, etc.

These properties are updated in bulk, once a day. The data gets populated into tabs in DoorDash's data warehouse after a nightly run of the respective ETL jobs.

The CDC patterns described for Human Operator Changes don't work here because you don't constantly have changes/updates through the day. Instead, you have one bulk update that happens once a day.

Using the CDC pattern described above would overwhelm the system when making the bulk update due to the size of the update.

Therefore, DoorDash engineers built a custom Flink source function which spreads out the ETL ingestion over a 24 hour interval so that the systems don't get overwhelmed.

The Flink source function will periodically stream rows from an ETL table to Kafka in batches, where the batch size is chosen to ensure that the downstream systems do not get overwhelmed.

Sending documents to Elasticsearch

Once the Assembler application publishes data to Kafka, the consumer (ES Sink) will read those messages, transform them according to the specific index schema, and then send them to their appropriate index in Elasticsearch.

ES Sink utilizes Flink Elasticsearch Connector to write JSON documents to Elasticsearch.

It has rate limiting and throttling capabilities out of the box, which are essential for protecting Elasticsearch clusters when the system is under heavy write load.

Results

With the new search indexing platform, updates happen much faster. The time needed to reindex existing stores and items on the platform fell from 1 week to 2 hours.

The reliance on open source tools for the index means a lot of accessible documentation online and engineers with this expertise who can join the DoorDash team in the future.

For information on how DoorDash backfilled the search index (and more!), read the full blog [post here](#).

The Architecture of Databases

Alex Petrov is a software engineer at Apple. He wrote an amazing book on Database Internals, where he does a deep dive on how distributed data systems work. I'd highly recommend you pick up a copy if you want to learn more on the topic.

We'll be summarizing a small snippet from his book on the architecture behind Database Management Systems (DBMSs).

Summary

You've probably used a DBMS like Postgres, MySQL, etc.

They provide an extremely useful abstraction that you can use in your applications for storing and (later) retrieving data.

All DBMSs provide an API that you can use through some type of Query Language to store and retrieve data.

They also provide a set of guarantees around *how* this data is stored/retrieved. A couple examples of such guarantees are

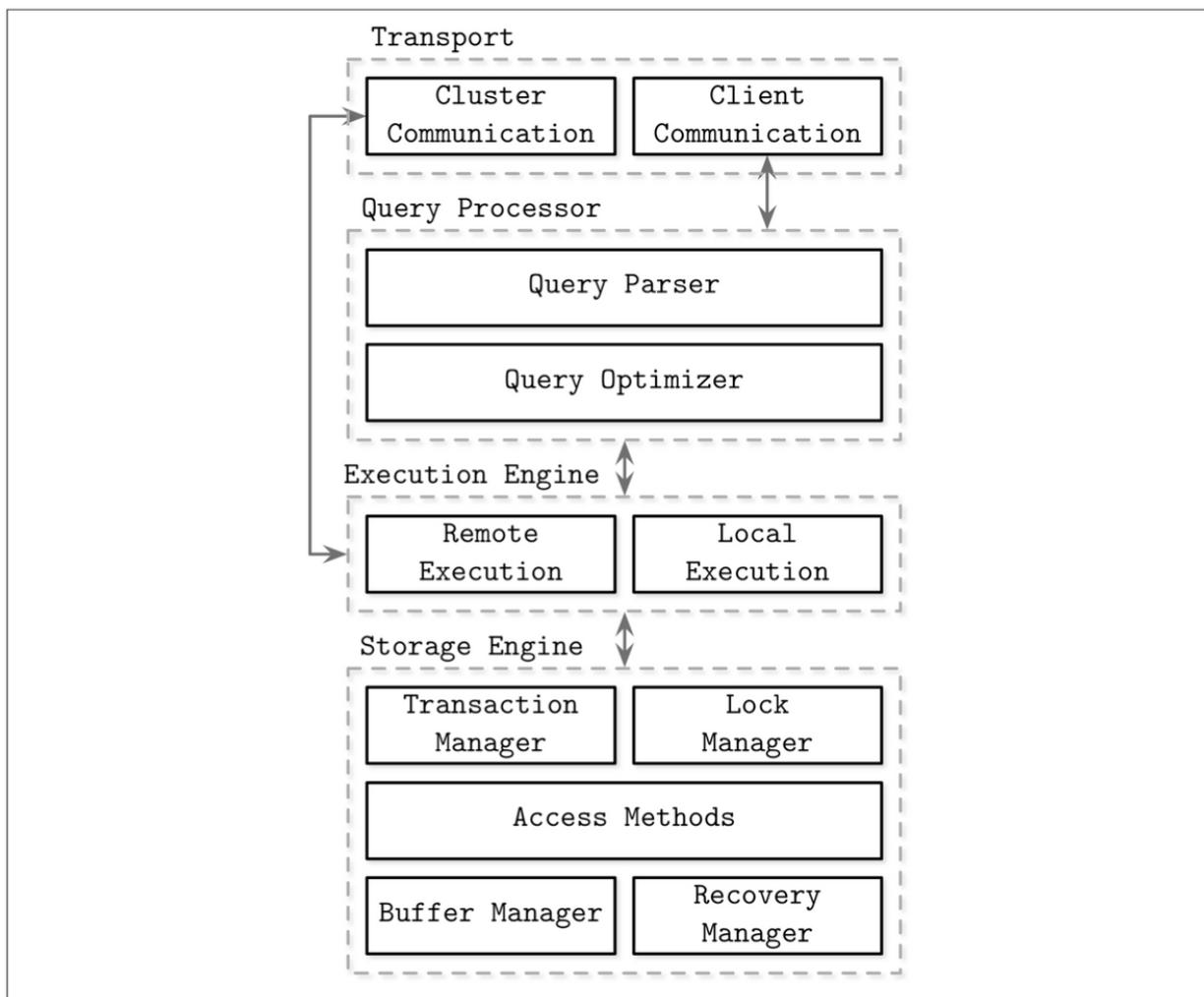
- Durability - guarantees that you won't lose *any* data if the DBMS crashes
- Consistency - After you write data, will all subsequent reads always give the most recent value of the data? (*this is important for distributed databases*)
- Read/Write Speeds - [IOPS](#) is the standard measure of input and output operations per second on storage devices.

The architecture of the various Database Management Systems vary widely based on their guarantees and design goals. A database designed for [OLTP](#) use will be designed differently than a database meant for [OLAP](#).

An in-memory DBMS (designed to store data primarily in memory and use disk for recovery and logging) will also be designed differently than a disk-based DBMS (designed to primarily store data on disk and use memory for caching).

However, databases have some common themes in their various architectures and knowing these themes can provide a useful model for how they work.

The general architecture can be described by this diagram.



Database Management Systems use a client/server model. Your application is the client and the DBMS is the server (either hosted on the same machine or on a different machine).

The *Transport* system is how the DBMS accepts client requests.

Client requests come in the form of a database query and are usually expressed in some type of a query language (ex. SQL).

After receiving the query, the Transport system will pass the query on to the Query Processor.

The Query Processor will first parse the query (using an Abstract Syntax Tree for ex.) and make sure it is valid.

Checking for validity means making sure that the query makes sense (all the commands are recognized, the data accessed is valid, etc.) and also that the client is correctly permissioned to access/modify the data that they're requesting.

If the query isn't valid, then the database will return an error to the client.

Otherwise, the parsed query is passed to the [Query Optimizer](#).

The optimizer will first eliminate redundant parts of the query and then use internal database statistics (index cardinality, approximate intersection size, etc.) to find the most efficient way to execute the query.

For distributed databases, the optimizer will also consider data placement like which node in the cluster holds the data and the costs associated with the transfer.

The output from the Optimizer is an [Execution Plan](#) that describes the optimal method of executing the query. This plan is also called the Query Plan or Query Execution Plan.

This execution plan gets passed on to the Execution Engine which carries out the plan.

When you're using a distributed database, the execution plan can involve remote execution (making network requests for data that is stored on a different machine).

Otherwise, it's just local execution (carrying out queries for data that is stored locally).

Remote execution involves *Cluster Communication*, where the DBMS communicates with other machines in the database cluster and sends them requests for data. As you can see in the diagram above, that's part of the Transport layer.

Local execution involves talking to the *Storage Engine* to get the data.

The Storage Engine is the component in the database that is directly responsible for storing, retrieving and managing data in memory and on disk.

Storage Engines typically provide a simple data manipulation API (allowing for CRUD features) and contain all the logic for the actual details of how to manipulate the data.

Examples of Storage Engines include [BerkeleyDB](#), [LevelDB](#), [RocksDB](#), etc.

Databases will often allow you to pick the Storage Engine that's being used.

MySQL, for example, has several choices for the storage engine, including RocksDB and InnoDB.

The Storage Engine consists of several components

Transaction Manager - responsible for creating transaction objects and managing their atomicity (either the entire transaction succeeds or it is rolled back).

Lock Manager - Transactions will be executing concurrently, so the Lock Manager manages the locks on database objects being accessed by each transaction (and releasing those locks when the transaction either commits or rolls back).

Access Methods - These manage access, compression and organizing data on disk.

Access methods include heap files and storage structures such as B-trees.

Buffer Manager - This manager caches data pages in RAM to reduce the number of accesses to disk.

Recovery Manager - Maintains the operation log and restoring the system state in case of a failure.

Different Storage Engines make different tradeoffs between these components resulting in differing performance for things like compression, scaling, partitioning, speed, etc.

For more, be sure to check out Database Internals by Alex Petrov.

Observability at Twitter

Previously, Twitter used a home-grown logging system called Loglens. Engineers had quite a bit of trouble with Loglens, mainly around its low ingestion capacity and limited query capabilities.

To fix this, Twitter adopted Splunk for their logging system. After the switch, they've been able to ingest 4-5 times more logging data with faster queries.

Kristopher Kirland is a senior Site Reliability Engineer at Twitter and he wrote a great [blog post](#) on this migration and some of the challenges involved (*published August 2021*).

Here's a summary

The Legacy Logging System

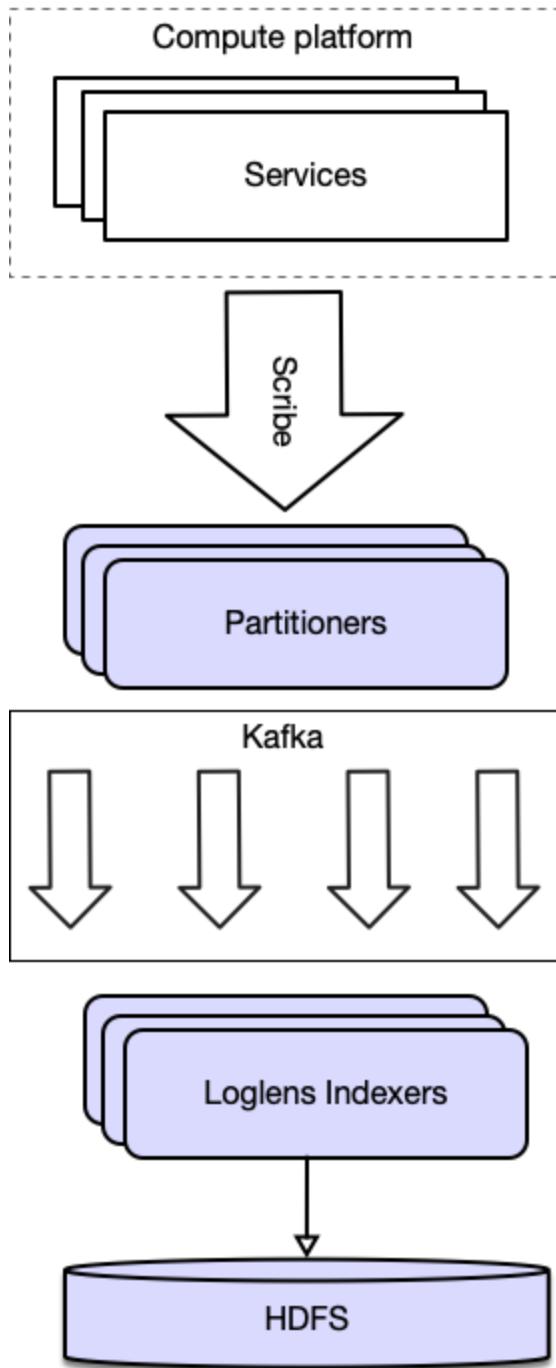
Before Loglens (the legacy centralized logging system), Twitter engineers had great difficulty with browsing through the different log files from their various backend services. This would be super frustrating when engineers were investigating an ongoing incident.

To solve this, they designed Loglens as a centralized logging platform that would ingest logs from all the various services that made up Twitter.

Their goals for this platform were

- Ease of onboarding
- Low cost
- Little time investment from developers for ongoing maintenance and improvements

Log files would be written to local [Scribe](#) daemons, forwarded onto Kafka and then ingested into the Loglens indexing system and written to [HDFS](#).



While running Loglens as their logging backend, the platform ingested around 600k events per second per datacenter.

However, only 10% of the logs ingested were actually submitted. The other 90% were discarded by the rate limiter to avoid overwhelming Loglens.

The logging system had very little resource investment (little investment was one of the goals) and that led to a poor developer experience with the platform.

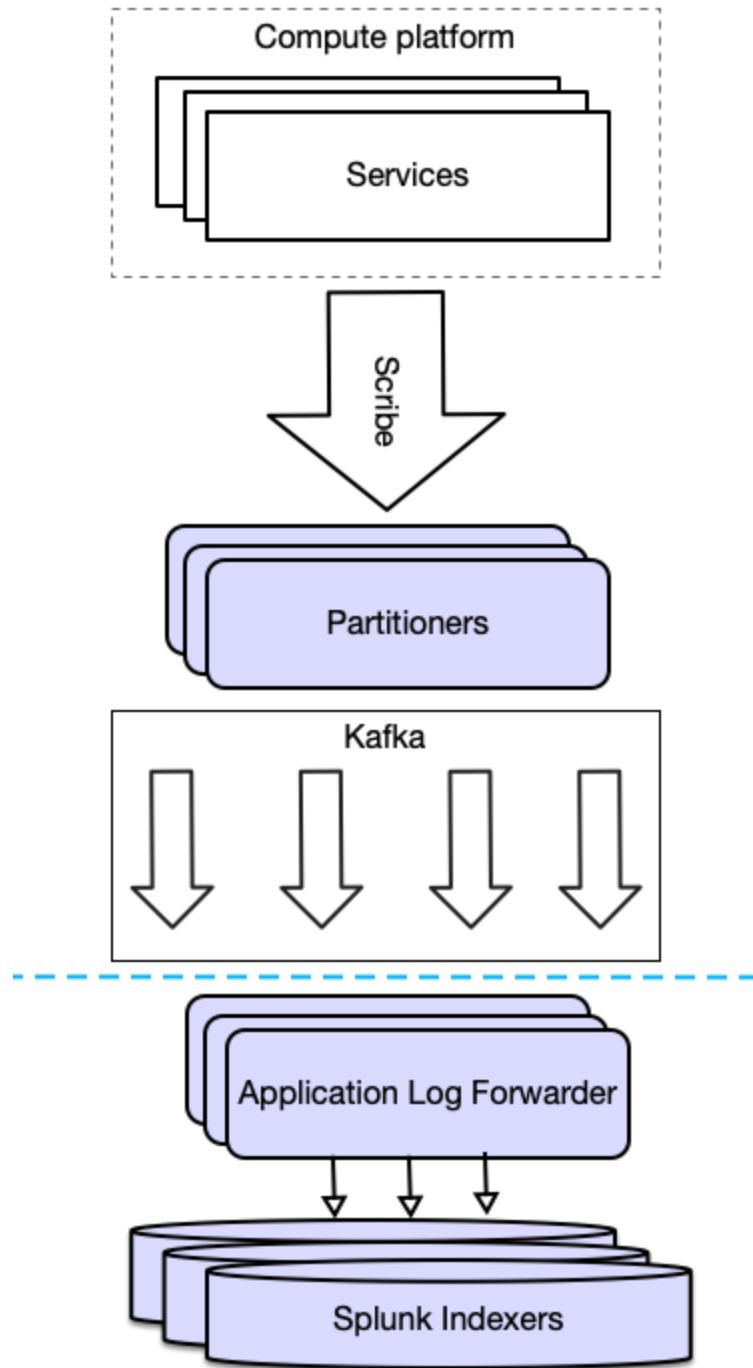
Transitioning to Splunk

Twitter engineers decided to switch to Splunk for their centralized logging platform.

They chose Splunk because it could scale to their needs; ingesting logs from hundreds of thousands of servers in the Twitter fleet. Splunk also offers flexible tooling that satisfies the majority of Twitter's log analysis needs.

Due to the loosely coupled design of Loglens, migrating to Splunk was pretty straightforward.

Twitter engineers created a new service to subscribe to the Kafka topic that was already in use for Loglens, and then they forwarded those logs to Splunk. The new service is called the Application Log Forwarder (ALF).



Twitter uses this set up for the majority of their logs (over 85%) but they also use the Splunk Universal Forwarder. With the Universal Forwarder, they just install that application on each server in their fleet and it starts ingesting logs.

With Splunk, Twitter now has a much greater ingestion capacity compared to Loglens. As of August 2021, they collect nearly 42 terabytes of data per datacenter each day.

That's 5 million events per second per datacenter, which is far greater than what Twitter was able to do with Loglens.

They also gained some other features like greater configurability, the ability to save and schedule searches, complex alerting, a more robust and flexible query language, and more.

Challenges of running Splunk

Some of the challenges that Twitter engineers faced with the migration were

- Control of flow of data is limited - As stated previously, Twitter uses the Application Log Forwarder (ALF) for 85% of their logging and the Splunk Universal Forwarder for 15% of logging. If something goes wrong with a service and they start to flood the system with enough logging events to threaten the stability of the Splunk Enterprise Clusters, the ALF can rate limit by log priority or originating service. However, the Splunk Universal Forwarder lacks the flexibility that the ALF has.
- Server maintenance in large clusters is a pain - Server maintenance, like regular reboots, presents a significant challenge. Rebooting too many servers at once can cause interruptions for Splunk searches and poor ingestion rates of new logging data.
- Managing Configuration - Twitter uses the typical configuration management tools like Puppet/Chef for the majority of needs, but they fell short of what they wanted for managing indexes and access controls. They had to create their own service that generated specific configuration files and deployed them to the correct servers.
- Modular Inputs are not Resilient - One feature of Splunk is Splunkbase, where you can find a large array of add-ons for all the various products you might be using. Many add-ons give you the ability to collect data from the API of third-party applications like GitHub, Slack, Jenkins, etc. However, Twitter engineers found that most of these add-ons are not designed to run in a

resilient manner. Instead, they implemented their own plugins to run on their compute infrastructure.

For more information, check out the full blog post [here](#).

Language Implementations Explained

Bob Nystrom is a software engineer at Google, where he works on the Dart Programming language.

He wrote an *amazing* book called *Crafting Interpreters*, where he walks you through how programming language implementations work. In the book, you'll build two interpreters, one in Java and another in C.

He published the entire book for free [here](#), but I'd highly suggest you [support the author](#) if you have the means to do so.

His section on *A Map of the Territory* gives a fantastic introduction to programming language implementations, so here's a summary of that section (with my own commentary added in).

Languages vs. Language Implementations

First of all, it's important to note that a programming language *implementation* is different from a programming language.

The programming language refers to the syntax, keywords, etc. The programming language itself is usually designed by a committee and there are some standard documents that describe the language. These documents are usually called the [Programming Language Specification](#).

(Not all languages have a specification. Python, for example, has the [Python Language Reference](#), which is the closest thing to its specification.)

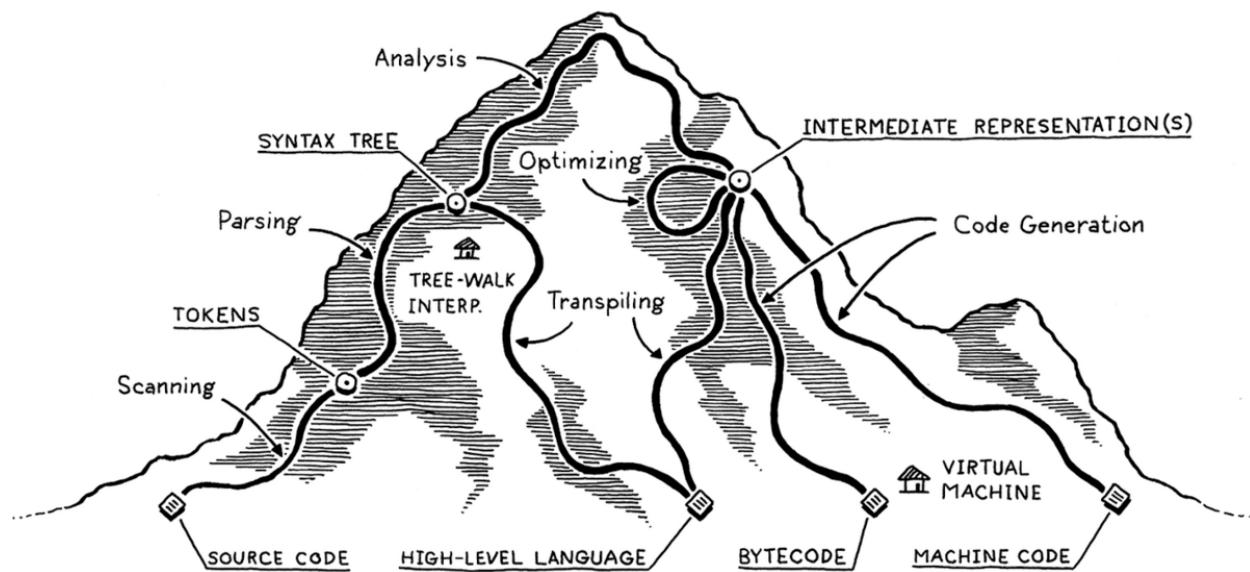
The language implementation is the actual software that allows you to run code from that programming language. Typically, an implementation consists of a compiler/interpreter.

A programming language can have *many* different language implementations, and these implementations can all be quite different from each other. The key factor is that all

these implementations must be able to run code that is defined according to the programming language specification.

The most popular implementation for Python is CPython but there's also PyPy (JIT compiler), Jython (Python running on the JVM), IronPython (Python running on .NET) and *many more* [implementations](#).

The Architecture of a Language Implementation



Language implementations are obviously built differently, but there are some general patterns.

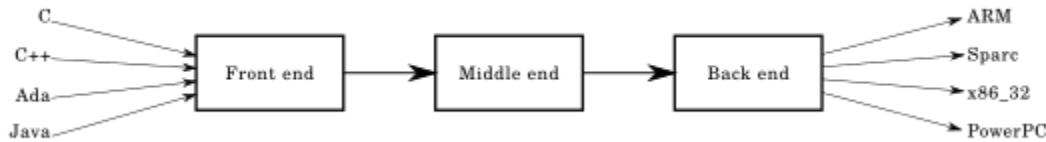
A language implementation can be subdivided into the following parts

- Front end - takes in your source code and turns it into an [intermediate representation](#).
- Middle end - takes the intermediate representation and applies optimizations to it.
- Back end - takes the optimized intermediate representation and turns it into machine code or bytecode.

The intermediate representation is a data structure or code that makes the compiler more modular. It allows you to reuse your front end for multiple target platforms and reuse your backend for multiple source languages.

For example, you can write multiple back ends that turn the intermediate representation into machine code for x86, ARM, and other platforms and then reuse the same front end that turns your C code into intermediate representation.

LLVM is designed around a high-level assembly language that is named “intermediate representation” ([here’s the language reference for LLVM IR](#)) while CPython uses a data structure called the [Control Flow Graph](#).



We’ll break down all of these concepts further...

Front end

As we said before, the front end is responsible for taking in your source code and turning it into an intermediate representation.

The first part is [scanning](#) (also known as lexical analysis). This is where the front end reads your source code and converts it into a series of tokens. A token is a single element of a programming language. It can be a single character, like a {, or it can be a word, like System.out.println.

After scanning and converting your source code into tokens, the next step is parsing.

A parser will take in the flat sequence of tokens and build a tree structure based on the programming language’s [grammar](#).

This tree is usually called an abstract syntax tree (AST). While the parser is creating the abstract syntax tree, it will let you know if there are any syntax errors in your code.

The front end will also handle tasks like binding. This is where every identifier (variable, etc.) gets linked to where it's defined.

If the language is statically typed, this is where type checking happens and type errors are reported.

In terms of storing this information, language implementations will do this differently.

Some will store it on the abstract syntax tree as attributes. Others will store it in a lookup table called a symbol table.

All this information will be stored in some type of intermediate representation.

There are a couple of well established styles of intermediate representation out there.

Examples include

- [Control Flow Graph](#)
- [Static Single Assignment](#)
- [Three-address Code](#)
- [Continuation-passing Style](#)

Having a shared intermediate representation helps make your compiler design much more modular.

Middle End

The middle end is responsible for performing various optimizations on the intermediate representation.

These optimizations are independent of the platform that's being targeted, therefore they'll speed up your code regardless of what the backend does.

An example of an optimization is [constant folding](#): if some expression always evaluates to the exact same value, then do that evaluation at compile time and replace the code for the expression with the result.

So, replace

```
pennyArea = 3.14159 * (0.75 / 2) * (0.75 / 2);
```

With the result of that expression.

```
pennyArea = 0.4417860938;
```

Other examples are removal of unreachable code ([reachability analysis](#)) and code that does not affect the program results ([dead code elimination](#)).

Back End

The back end is responsible for taking the optimized intermediate representation from the middle end and generating the machine code for the specific CPU architecture of the computer (or generating bytecode).

The back end may perform more analysis, transformations and optimizations that are specific for that CPU architecture.

If the back end produces bytecode, then you'll also need another compiler for each target architecture that turns that bytecode into machine code.

Or, many runtimes rely on a virtual machine, where a program emulates a hypothetical chip. The bytecode is run on that virtual machine.

An example is the Java Virtual Machine, which runs Java bytecode. You can reuse that backend and write frontends to handle different languages. Python, Kotlin, Clojure and Scala are a few examples of languages that have front ends that can convert that language into Java bytecode.

Airbnb's Architecture

Jessica Tai is an engineering manager at Airbnb where she works on platform infrastructure. She gave a great [talk at QCon](#) on Airbnb's architecture and how it's shifted over the years.

Here's a summary

Airbnb has been through three major stages in their architecture since the company's founding.

Airbnb started as a Ruby on Rails monolith, then transitioned to a microservices architecture and has now migrated to a hybrid between micro and macroservices (a macroservice aggregates multiple microservices).

We'll go through each architecture and talk about the pros/cons and why Airbnb migrated.

Monolith (2008 - 2017)

Airbnb started off with a Ruby on Rails monolith and it worked really well for the company.

Most engineers were full stack and could work on every part of the codebase, executing on end-to-end features by themselves. Features could be completed within a single team, which helped the company build new products very quickly.

However, as Airbnb entered hypergrowth, the number of engineers, teams, and lines of code scaled up very quickly.

It became impossible for a single engineer/team to have context on the entire codebase, so ownership and team boundaries were needed.

Airbnb struggled with drawing these team boundaries since the monolith was very tightly coupled. Code changes in one team were having unintended consequences for another team and who owned what was confusing for different parts of the codebase.

There were other scaling pain points like extremely slow deploys, sometimes taking over a day to get a single deploy done.

These issues were leading to a slower developer velocity and Airbnb decided to shift to a microservices oriented approach to reduce these pain points.

Microservices (2017 - 2020)

After learning from their experience with the monolith, Airbnb engineers wanted to be more disciplined with their approach to microservices.

They decided to have 4 different service types

1. Data fetching service to read and write data
2. A business logic service able to apply functions and combine different pieces of data together
3. A workflow service for write orchestration. This handles operations that involve touching pieces of data across multiple services.
4. A UI aggregation service that puts this all together for the UI

To avoid ownership issues seen with the monolith, each microservice would only have one owning team (and each team could own multiple services).

With these changes, Airbnb also changed the way engineering teams were structured.

Previously, engineering teams were full stack and able to handle anything. But now, with microservices, Airbnb shifted to teams that were just focused on a certain parts of the stack. Some were focused on certain data services while others were focused on specific pieces of business logic.

Airbnb also had a specific team that was tasked with running the migration of monolith to microservice. This team was responsible for building tooling to help with the

migration and to teach Airbnb engineers the best practices of microservice building and operations.

After a few years into the microservices migration, new challenges started to arise.

Managing all these services and their dependencies was quite difficult. Teams needed to be more aware of the service ecosystem to understand where any dependencies may lie.

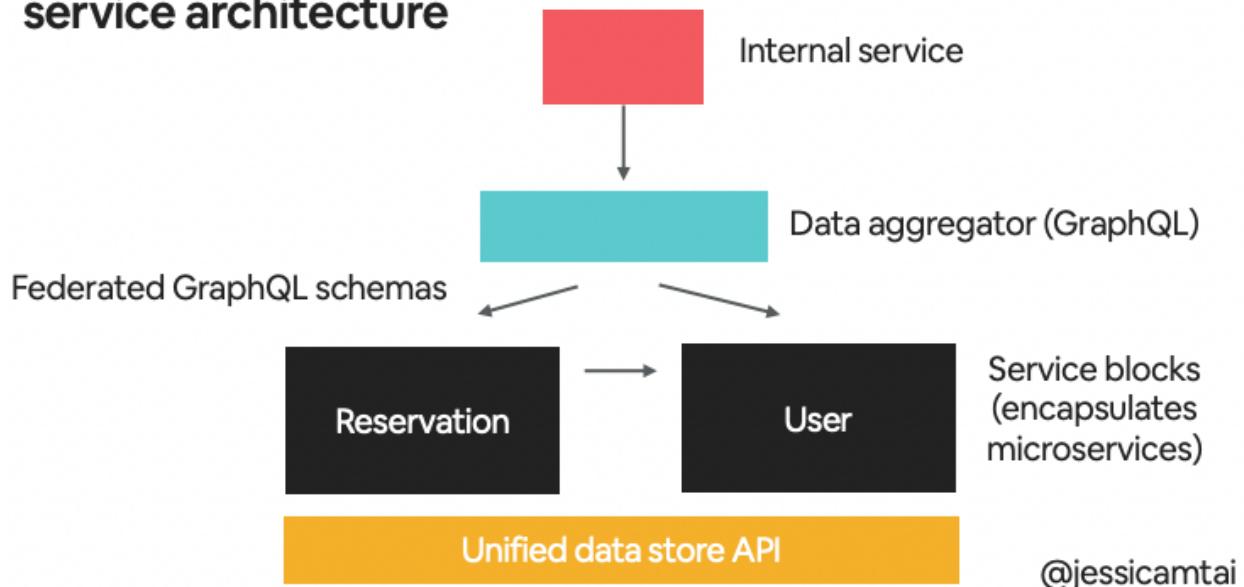
Building an end-to-end feature meant using various services across the stack, so different engineering teams would all need to be involved. All of these teams needed to have similar priorities around that feature, which was difficult to manage as each team owned multiple services.

Micro + Macroservices (2020 -)

To address those collaboration challenges, Airbnb is now instituting a hybrid approach between micro and macroservices. This model focuses on the unification of APIs and on consolidating to make clear places to go for certain pieces of data or functionality.

They're creating a system where their internal backend service gets its data from the data aggregation service. The data aggregator then communicates with the various service blocks where each service block encapsulates a collection of microservices.

Enforce paved path for micro+macro service architecture



A challenge that Airbnb engineers foresee with this approach is that the data aggregator can become a new monolith. To avoid that, they're very disciplined about what data/service belongs where.

For the service block layer, engineers need to make sure that they're defining the schema boundaries in a clean way. There are many pieces of data/logic that can span multiple entities, so it needs to be clearly defined.

For more details, you can watch Jessica's full talk [here](#).

The Architecture of Apache Spark

Spark is an open source project that makes it much easier to run computations on large, distributed data. It's widely used to run datacenter computations and its popularity has been exploding since 2012.

It's now become one of the most popular open source projects in the big data space and is used by companies like Amazon, Tencent, Shopify, eBay and more.

Before Spark, engineers relied on Hadoop MapReduce to run computations on their data, but there were quite a few issues with that approach.

Spark was introduced as a way to solve those pain points, and it's quickly evolved into much more.

We'll talk about why Spark was created, what makes Spark so fast and how it works under the hood.

We'll start with a brief overview of MapReduce.

History of MapReduce

In a previous tech dive, we talked about Google [MapReduce](#) and how Google was using it to run massive computations to help power Google Search.

MapReduce introduced a new parallel programming paradigm that made it much easier to run computations on massive amounts of distributed data.

Although Google's implementation of MapReduce was proprietary, it was re-implemented as part of [Apache Hadoop](#).

Hadoop gained widespread popularity as a set of open source tools for companies dealing with massive amounts of data.

How MapReduce Works

Let's say you have 100 terabytes of data split across 100 different machines. You want to run some computations on this data.

With MapReduce, you take your computation and split it into a Map function and a Reduce function.

You take the code from your map function and run it on each of the 100 machines in a parallel manner.

On each machine, the map function will take in that machine's chunk of the data and output the results of the map function.

The output will get written to local disk on that machine (or a nearby machine if there isn't enough space on local).

Then, the reduce function will take in the output of all the map functions and combine that to give the answer to your computation.

Issues with MapReduce

The MapReduce framework on Hadoop had some shortcomings that were becoming big issues for engineers.

- Iterative Jobs - A common use case for MapReduce is to stack multiple MapReduce jobs sequentially, and run them one after the other. MapReduce will write to disk after both the Map and Reduce steps, so this leads to a huge amount of disk I/O. Disk I/O can obviously be very slow, so this caused large MapReduce jobs (involving multiple MapReduce steps one after another) to be very slow and take hours/days.
- Interactive Analysis - When you store data on Hadoop (using HDFS), you'll want to run ad-hoc exploratory queries to better understand your data. Doing this with MapReduce can be a pain because of how unintuitive it can be to create Map and Reduce functions to do your data exploration. Instead, you'll

use something like Hive (an SQL query engine for Hadoop) so you can just write SQL queries to view your data. However, with Hadoop, Hive will be executing those SQL queries using MapReduce, which means significant latency for the reasons described above (lots of disk I/O).

- Lack of Flexibility - Hadoop MapReduce works for general batch processing tasks, but it becomes very unwieldy for handling other workloads like machine learning, streaming, or interactive SQL queries (described above). Turning complex jobs into Map and Reduce functions can be difficult. This meant that other tools had to be developed to handle those workloads like Hive, Storm, Mahout, etc.

Creation of Apache Spark

Apache Spark was created as a successor to MapReduce to ease these problems.

The main goal was to create a *fast* and *versatile* tool to handle distributed processing of large amounts of data. The tool should be able to handle a variety of different workloads, with a specific emphasis on workloads that reuse a working set of data across multiple operations.

Many common machine learning algorithms will repeatedly apply a function to the same dataset to optimize a parameter (ex. [Gradient descent](#)).

Running a bunch of random SQL queries on a dataset to get a feel for it is another example of reusing a working set of data across multiple operations (SQL queries in this scenario).

Spark is designed to handle these operations with ease.

Overview of Spark

Spark is a program for distributed data processing, so it runs on top of your data storage layer. You can use Spark on top of Hadoop Distributed File System, MongoDB, HBase, Cassandra, Amazon S3, RDBMSs and a bunch of other storage layers.

In a Spark program, you can [transform](#) your data in different ways (filter, map, intersection, union, etc.) and Spark can distribute these operations across multiple computers for parallel processing.

Spark offers nearly 100 high-level, commonly needed data processing operators and you can use Spark with Scala, Java, Python and R.

Spark also offers libraries on top to handle a diverse range of workloads.

- Spark SQL will let you use SQL queries to do data processing.
- Spark MLlib has common machine learning algorithms like logistic regression.
- Spark Structured Streaming lets you process real-time streaming data from something like Kafka or Kinesis.
- GraphX will let you manipulate graphs and offers algorithms for traversal, connections, etc. You can use algorithms like pagerank, triangle counting and connected components.

Why is Spark Fast?

Spark's speed comes from two main architectural choices

1. Lazy Evaluation - When you're manipulating your data, Spark *will not* execute your manipulations (called [transformations](#) in Spark lingo) immediately. Instead, Spark will take your transformations (like sort, join, map, filter, etc.) and keep track of them in a [Directed Acyclic Graph](#) (DAG). A DAG is just a graph (a set of nodes and edges) where the nodes have directed edges (the first transformation will point to the second transformation and so on) and the graph has no cycles. Then, when you want to get your results, you can trigger an [Action](#) in Spark. Actions trigger the evaluation of all the recorded transformations in the DAG. Because Spark knows what all your chained transformations are, Spark can then use its optimizer to construct the most efficient way to execute all the transformations in a parallel way. This helps make Spark much faster.
2. In Memory - We've said several times above that one of the issues with MapReduce is all the disk I/O. Spark solves this by retaining all the intermediate results in memory. After you trigger an Action, Spark will be calculating all the transformations in RAM using the memory from all the machines in your Spark cluster and then run the computations. If you don't have enough RAM, then Spark can also use disk and swap data between the two.

Architecture of Spark

As we said before, Spark is a distributed data processing engine that can process huge volumes of data distributed across thousands of machines.

The collection of machines is called a Spark cluster and the largest Spark cluster is around 8000 machines. (Note. You can also run Spark on a single machine. If you want, you can download it from the [Apache website](#))

Leader-Worker Architecture

Spark is based on a leader-worker architecture. In Spark lingo, the leader is called the Spark *driver* while the worker is called the Spark *executor*.

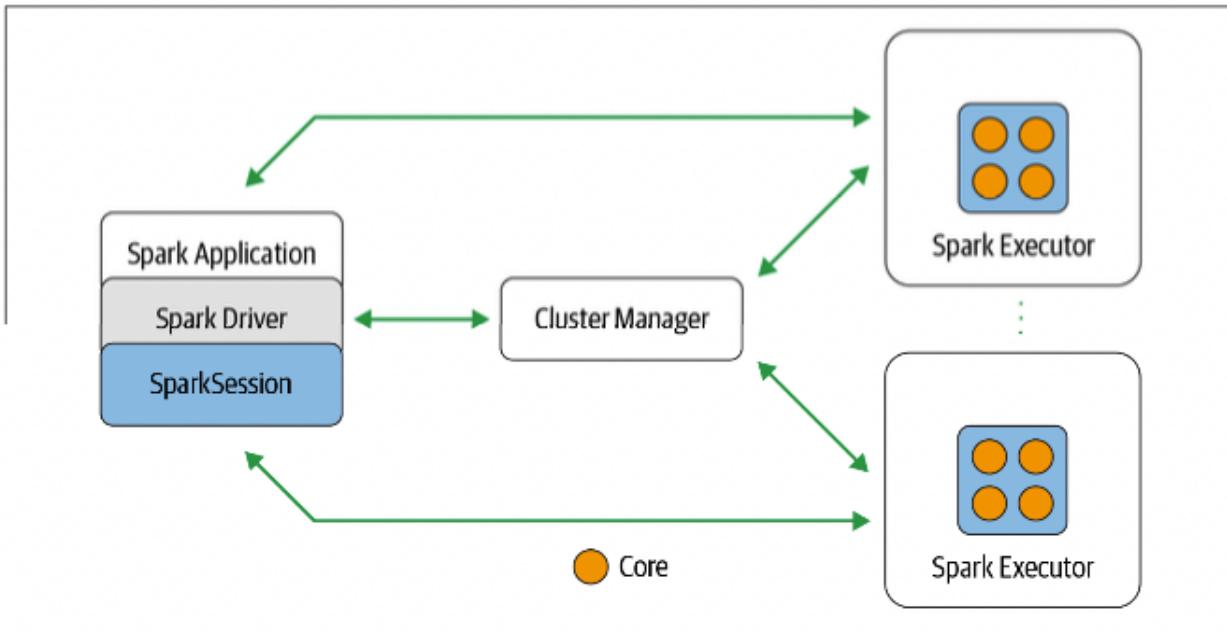
A Spark application has a single driver, where the driver functions as the central coordinator. You'll be interacting with the driver with your Scala/Python/R/Java code and you can run the driver on your own machine or on one of the machines in the Spark cluster.

The executors are the worker processes that execute the instructions given to them by the driver. Each Spark executor is a JVM process that is run on each of the nodes in the Spark cluster (you'll mostly have one executor per node).

The Spark executor will get assigned tasks that require working on a partition of the data that is closest to them in the cluster. This helps reduce network congestion.

When you're working with a distributed system, you'll typically use a [cluster manager](#) (like Apache Mesos, Kubernetes, Docker Swarm, etc.) to help manage all the nodes in your cluster.

Spark is no different. The Spark driver will work with a cluster manager to orchestrate the Spark Executors. You can configure Spark to use Apache Mesos, Kubernetes, Hadoop YARN or Spark's built-in cluster manager.



Resilient Distributed Dataset

When Spark runs your computations on the given datasets, it uses a data structure called a Resilient Distributed Dataset (RDD).

RDDs are the fundamental abstraction for representing data in Spark and they were first introduced in the [original Spark paper](#).

Spark will look at your dataset across all the partitions and create an RDD that represents it. This RDD will then be stored in memory where it will be manipulated through transformations and actions.

The key features of RDDs are

- Resilience - RDDs are fault-tolerant and able to survive failures of the nodes in the Spark cluster. As you call transformation operations on your RDD, Spark will be building up a DAG of all the transformations. This DAG can be used to track the [data lineage](#) of all the RDDs so you can reconstruct any of the past RDDs if one of the machines fails. Just note, this is fault tolerance for the RDD, *not* for the underlying data. Spark is assuming that the storage layer

(HDFS, S3, Cassandra, whatever) is handling redundancy for the underlying data.

- Distributed - Spark assumes your data is split across multiple machines so RDDs are also split across a cluster of machines. Spark will place executors close to the underlying data to reduce network congestion.
- Immutability - RDDs are immutable. When you apply transformations to an RDD, you don't change that RDD but instead create a new RDD. Immutability means every RDD is a deterministic function of the input. This makes caching, sharing and replication of RDDs much easier.

Directed Acyclic Graph

As you're running your transformations, Spark will *not* be executing any computations.

Instead, the Spark driver will be adding these transformations to a Directed Acyclic Graph. You can think of this as just a flowchart of all the transformations you're applying on the data.

Once you call an action, then the Spark driver will start computing all the transformations. Within the driver are the DAG Scheduler and the Task Scheduler. These two will manage executing the DAG.

When you call an action, the DAG will go to the DAG scheduler.

The DAG scheduler will divide the DAG into different stages where each stage contains various tasks related to your transformations.

The DAG scheduler will run various optimizations to make sure that the stages are being done in the most optimal way to eliminate any redundant computations. Then, it will create a set of stages and then pass this to the Task Scheduler.

The Task Scheduler will then coordinate with the Cluster Manager (Apache Mesos, Kubernetes, Hadoop YARN, etc.) to execute all the stages using the machines in your Spark cluster and get the results from the computations.

Netflix's Rapid Event Notification System

Netflix is an online video streaming service that operates at insane scale. They have more than 220 million active users and account for more of the world's downstream internet traffic than YouTube (in 2018, Netflix accounted for ~15% of the world's downstream traffic).

These 220 million active users are accessing their Netflix account from multiple devices, so Netflix engineers have to make sure that all the different clients that a user logs in from are synced.

You might start watching Breaking Bad on your iPhone and then switch over to your laptop. After you switch to your laptop, you expect Netflix to continue playback of the show exactly where you left off on your iPhone.

Syncing between all these devices for all of their users requires an immense amount of communication between Netflix's backend and all the various clients (iOS, Android, smart TV, web browser, Roku, etc.). At peak, it can be about 150,000 events per second.

To handle this, Netflix built RENO, their Rapid Event Notification System.

Ankush Gulati and David Gevorkyan are two senior software engineers at Netflix, and they wrote a great [blog post](#) on the design decision behind RENO.

Here's a Summary

Netflix users will be using their account with different devices.

Netflix engineers have to make sure that things like viewing activity, membership plan, movie recommendations, profile changes, etc. are synced between all these devices.

The company uses a microservices architecture for their backend, and built the RENO service to handle this task.

There were several key design decisions behind RENO

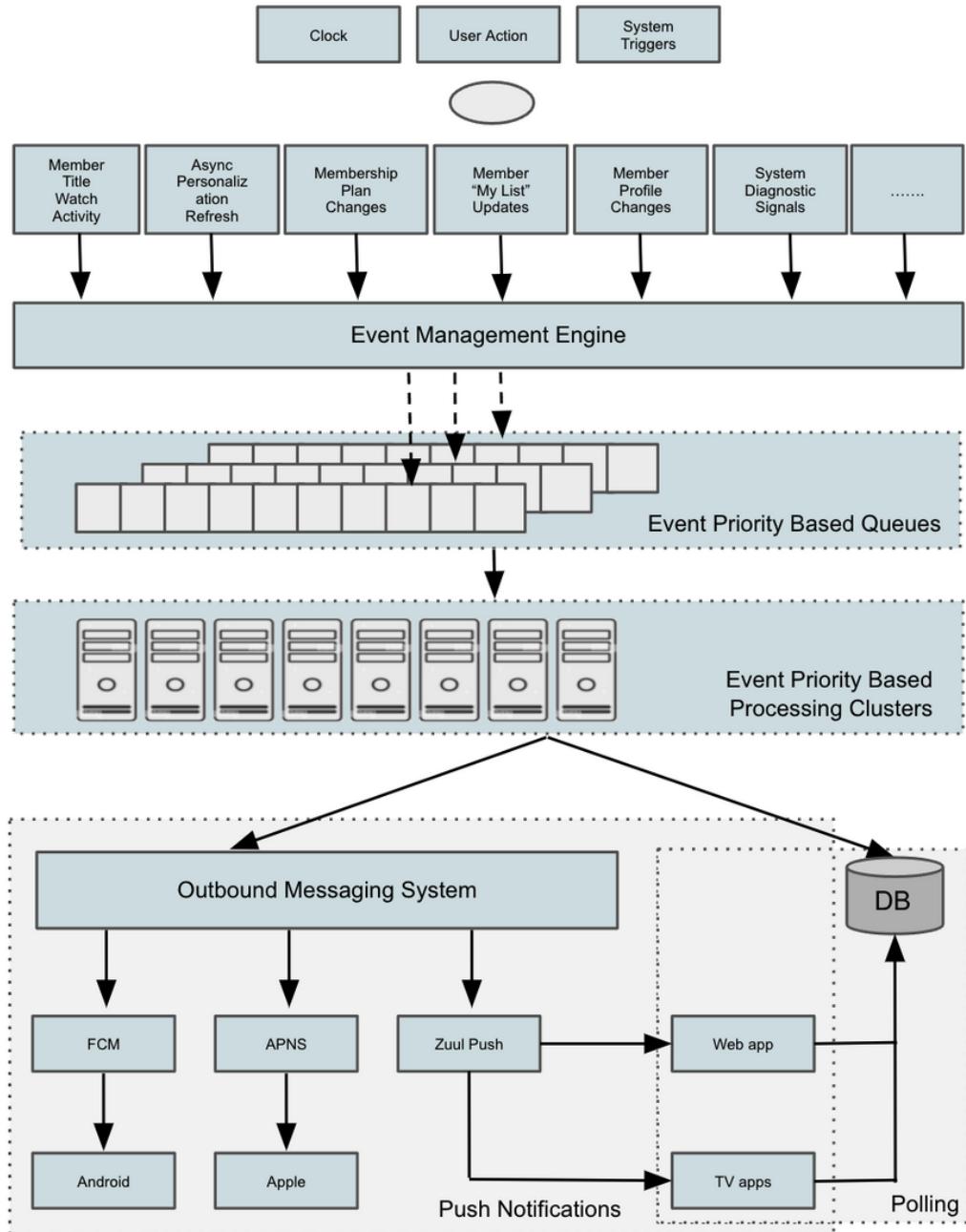
1. Single Event Source - All the various events (viewing activity, recommendations, etc.) that RENO has to track come from different internal systems. To simplify this, engineers used an Event Management Engine that serves as a level of indirection. This Event Management Engine layer is the single source of events for RENO. All the events from the various backend services go to the Event Management Engine, from where they're passed to RENO.
2. Event Prioritization - If a user changes their child's profile maturity level, that event change should have a very high priority compared to other events. Therefore, each event-type that RENO handles has a priority assigned to it and RENO then shards by that event priority. This way, Netflix can tune system configuration and scaling policies differently for events based on their priority.
3. Hybrid Communication Model - RENO has to support mobile devices, smart TVs, browsers, etc. While a mobile device is almost always connected to the internet and reachable, a smart TV is only online when in use. Therefore, RENO has to rely on a hybrid push AND pull communication model, where the server tries to deliver all notifications to all devices immediately using push. Devices will pull from the backend at various stages of the application lifecycle. Solely using pull doesn't work because it makes the mobile apps too chatty and solely using push doesn't work when a device is turned off.
4. Targeted Delivery - RENO has support for device specific notification delivery. If a certain notification only needs to go to mobile apps, RENO can solely deliver to those devices. This limits the outgoing traffic footprint significantly.
5. Managing High RPS - At peak times, RENO serves 150,000 events per second. This high load can put strain on the downstream services. Netflix handles this high load by adding various gate checks before sending an event. Some of the gate checks are

1. Staleness - Many events are time sensitive so RENO will not send an event if it's older than its staleness threshold
2. Online Devices - RENO keeps track of which devices are currently online using [Zuul](#). It will only push events to a device if it's online.
3. Duplication - RENO checks for any duplicate incoming events and corrects that.

Architecture

Here's a diagram of RENO.

We'll go through all the components below.



At the top, you have Event Triggers.

These are from the various backend services that handle things like movie recommendations, profile changes, watch activity, etc.

Whenever there are any changes, an event is created. These events go to the Event Management Engine.

The Event Management Engine serves as a layer of indirection so that RENO has a single source of events.

From there, the events get passed down to Amazon SQS queues. These queues are sharded based on event priority.

AWS Instance Clusters will subscribe to the various queues and then process the events off those queues. They will generate actionable notifications for all the devices.

These notifications then get sent to Netflix's outbound messaging system. This system handles delivery to all the various devices.

The notifications will also get sent to a Cassandra database. When devices need to pull for notifications, they can do so using the Cassandra database (remember it's a Hybrid Communications Model of push and pull).

The RENO system has served Netflix well as they've scaled. It is horizontally scalable due to the decision of sharding by event priority and adding more machines to the processing cluster layer.

For more details, you can read the full blog post [here](#).

Bloom Filters

A bloom filter is a very space-efficient data structure that can quickly (*constant time*) tell you if an item doesn't exist in a group of items.

When you're creating a new username on Twitter and they have to check whether that username is already taken, a bloom filter is a great data structure to use for speeding that up.

How Bloom Filters Work

A bloom filter is like a simpler version of a hash table.

The core data structure in a bloom filter is a [bit vector](#) (an array of bits). This bit vector will be used to keep track of items that exist in the set.

When you want to insert an item, you'll first use a [hash function](#) (or multiple hash functions) to hash that item into an integer.

Then, you can mod that integer by the number of slots in your bit vector

```
integer % num_slots_bit_vector
```

This will give you a slot in your bit vector for that item. You set that slot's bit to 1. Then, you can add the item to your database (or whatever storage layer you're using).

If you want to check if some item exists in your bloom filter, you can repeat the process of hashing and modulus to find the corresponding slot in the bit vector. If the slot in the bit vector is not set to 1, then you immediately know that the item does not exist in the set. You don't have to query your database (which is a lot more expensive than using the bloom filter).

However, if the slot in the bit vector for that item is set to 1, then that doesn't tell you for sure that the item exists in the set. You'll have to do a further check within your database to know for certain.

The bloom filter will only tell you if an item *doesn't* exist in the set.

This is because of possible collisions with your hash function ([pigeonhole principle](#)). Your bit vector only has a limited number of slots, and the number of possible items is much, much larger than the number of slots.

You'll eventually run into a scenario where two different items get mapped to the same slot in your bloom filter's bit vector.

Therefore, bloom filters will tell you if an item is either

1. possibly in the set
2. definitely not in the set

False positive matches are possible, but false negatives are not.

Comparing Bloom Filters and Hash Tables

Bloom filters are very similar to hash tables, but a hash table eliminates the possibility of false positive matches through [collision resolution](#).

The hash table solves the collision problem with solutions like [open addressing](#).

The downside of this is that it makes the hash table take up far more space than the bloom filter.

If you want to keep track of every single twitter username in your data structure, a hash table may become too large to store in-memory. In that situation, you'll want to use a bloom filter.

Example Use Case

Going back to the twitter example, let's say you're an engineer at twitter and you're working on the sign up form for new users.

When a new user tries to create their username, they need to quickly find out if their username has already been taken.

Sending a query to your database every single time a user attempts to create a username can result in unnecessary load on the database.

Therefore, you can use a bloom filter to quickly check if a username is unique.

If the username doesn't exist in the bloom filter, then you can avoid querying the database.

If the username does exist in the bloom filter, then you can query the database to check if the username is already taken or if the bloom filter match was a false positive.

Real World Use Cases

Databases use bloom filters extensively to reduce disk lookups for non-existent rows/columns. Apache Cassandra, Postgres and Google Big Table are just a few examples of databases that use bloom filters to reduce disk lookups.

The [Akamai](#) Content Delivery Network (one of the largest CDNs in the world) uses bloom filters to avoid “one-hit wonders” from being stored in their caches.

One-hit-wonders are objects that are requested by users just once. [Akamai uses a bloom filter](#) to detect the second request for a web object and then cache it only after the second request.

Scaling an API with Rate Limiters

Stripe Engineering wrote a fantastic [blog post](#) on how they think about rate limiters.

Here's a summary.

[Rate Limiting](#) is a technique used to limit the amount of requests a client can send to your server.

It's incredibly important to prevent DoS attacks from clients that are (accidentally or maliciously) flooding your server with requests.

A rule of thumb for when you should use a rate limiter is *if your users can reduce the frequency of their API requests without affecting the outcome of their requests, then a rate limiter is appropriate.*

For example, if you're running Facebook's API and you have a user sending 60 requests a minute to query for their list of Facebook friends, you can rate limit them without affecting their outcome. It's unlikely that they're adding new Facebook friends every single second.

Rate Limiting is great for day-to-day operations, but you'll occasionally have incidents where some component of your system is down and you can't process requests at your normal level.

In these scenarios, [Load Shedding](#) is a technique where you drop low-priority requests to make sure that critical requests get through.

Stripe is a payment processing company (you can use their API to collect payments from your users) so a critical request for them is a request to charge a user money.

An example of a non-critical method would be a request to read charge data from the past.

Stripe uses 4 different types of limiters in production (2 rate limiters and 2 load shedders).

Request Rate Limiter

Restricts each user to n requests per second. However, they also built in the ability for a user to briefly burst above the cap to handle legitimate spikes in usage.

Concurrent Requests Limiter

Restricts each user to n API requests in progress *at the same time*.

This helps stripe manage the load of their CPU-intensive API endpoints.

Fleet Usage Load Shedder

Stripe divides their traffic into two types: critical API methods and non-critical methods.

An example of a critical method would be creating a charge (charging a customer for something), while a non-critical method is listing a charge (looking at past charges).

Stripe always reserves a fraction of their infrastructure for critical requests. If the reservation number is 10%, then any non-critical request over the 90% allocation would be rejected with a 503 status code.

Worker Utilization Load Shedder

Stripe uses a set of workers to independently respond to incoming requests in parallel. If workers start getting backed up with requests, then this load shedder will shed lower priority traffic.

Stripe divides their traffic into 4 categories

- Critical Methods
- POSTs
- GETs
- Test mode traffic (traffic from developers testing the API and making sure payments are properly processed)

If worker capacity goes below a certain threshold, Stripe will begin shedding less-critical requests, starting from test mode traffic.

Building a rate limiter in practice

There are quite a few algorithms you can use to build a rate limiter. Algorithms include

Token Bucket - Every user gets a bucket with a certain amount of “tokens”. On each request, tokens are removed from the bucket. If the bucket is empty, then the request is rejected.

New tokens are added to the bucket at a certain threshold (every n seconds). The bucket can hold a certain number of tokens, so if the bucket is full of tokens then no new tokens will be added.

Fixed Window - The rate limiter uses a window size of n seconds for a user. Each incoming request from the user will increment the counter for the window. If the counter exceeds a certain threshold, then requests will be discarded.

After the n second window passes, a new window is created.

Sliding Log - The rate limiter tracks every user’s request in a time-stamped log. When a new request comes in, the system calculates the sum of logs to determine the request rate. If the request rate exceeds a certain threshold, then it is denied.

After a certain period of time, previous requests are discarded from the log.

Stripe uses the token bucket algorithm to do their rate limiting.

Serving Feature Data at Scale

Lyft uses machine learning extensively throughout their app. They use ML models to decide the optimal way to match drivers with riders, figure out the price for a ride, distribute coupons to riders, and a lot more.

In order for the ML models to run, Lyft engineers have to make sure the model's [features](#) are always available.

The features are the input that an ML model uses in order to get its prediction. If you're building a machine learning algorithm that predicts a house's sale price, some features might be the number of bedrooms, square footage, zip code, etc.

A core part of Lyft's Machine Learning Platform is their Feature Serving service, which makes sure that ML models can get low latency access to feature data.

Vinay Kakade worked on Lyft's Machine Learning Platform and he wrote a great [blog post](#) on the architecture of Lyft's Feature Serving service.

Here's a summary

Lyft's ML models are computed in two ways.

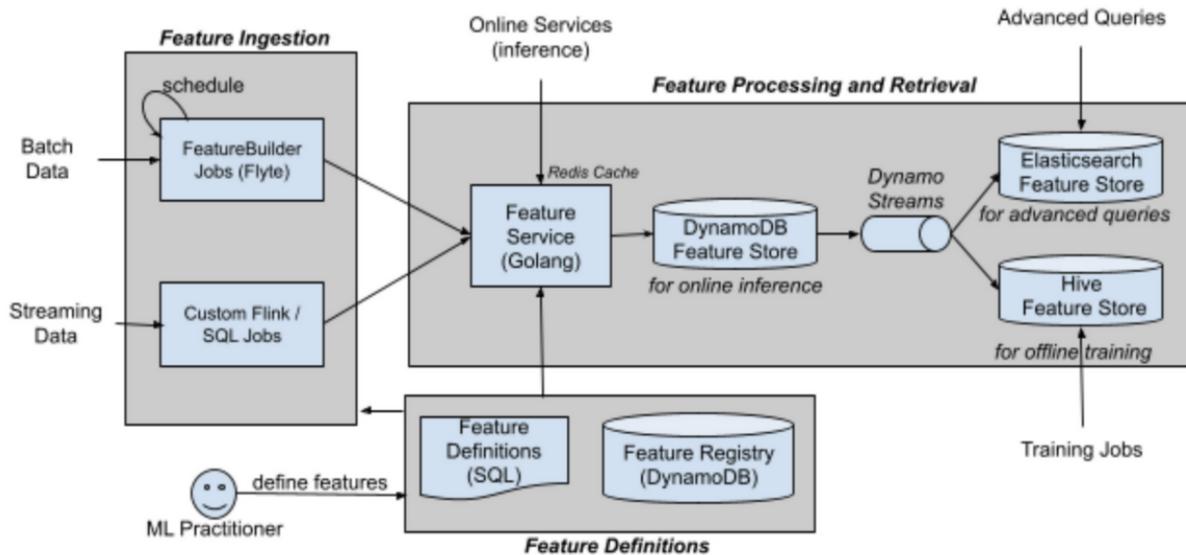
- Some are computed via batch jobs. Deciding which users should get a 10% off discount can be computed via a batch job that can run nightly.
- Others are computed in real time. When a user inputs her destination into the app, the ML model has to immediately output the optimal price for the ride.

Lyft also needs to train their ML models (determine the optimal model parameters to produce the best predictions) which is done via batch jobs.

The Feature Serving service at Lyft is responsible for making sure all features are available for both training ML models and for making predictions in production.

The service hosts several thousand features and serves millions of requests per minute with single-digit millisecond latency. It has 99.99%+ availability.

Here's the architecture.



The core parts of the Feature Serving service are the

- Feature Definitions
- Feature Ingestion
- Feature Processing & Retrieval

Feature Definitions

The features are defined in SQL. The complexity of the definitions can range from a single query to thousands of lines of SQL comprising complex joins and transformations.

The definitions also have metadata in JSON that describes the feature version, owner, validation information, and more.

Feature Data Ingestion

For features defined on batch data, Lyft uses [Flyte](#) to run regularly scheduled feature extraction jobs. The job executes SQL against Lyft's data warehouse and then writes to the Feature Service.

For real time feature data, Lyft uses [Apache Flink](#). They execute SQL against a [stream window](#) and then write to the Feature Service.

Feature Processing and Retrieval

The Feature Serving service is written in Golang and has gRPC and REST endpoints for writing and reading feature data.

When feature data is added to the service, it is written in both DynamoDB and Redis (Redis is used as a [write-through](#) cache to reduce read load on DynamoDB).

Lyft uses Dynamo streams to replicate the feature data to Apache [Hive](#) (their data warehouse tool) and Elasticsearch.

The Feature Serving service will then utilize the Redis cache, DynamoDB, Hive and Elasticsearch to serve requests for feature data.

For real-time ML models that need feature data back quickly, the Feature Serving service will try to retrieve the feature data from the Redis cache. If there is a cache miss, then it will retrieve the data from DynamoDB.

For batch-job ML models, they can retrieve the feature data from Hive. If they have an advanced query then they can also use Elasticsearch. You can read more about how Lyft uses Elasticsearch (and performance optimizations they've made) [here](#).

For more details on their Feature Serving service, you can read the full article [here](#).

Etsy's Journey to TypeScript

Salem Hilal is a software engineer on Etsy's Web Platform team. He wrote a great [blog post](#) on the steps Etsy took to adopt TypeScript.

Here's a summary

Etsy's codebase is a [monorepo](#) with over 17,000 JavaScript files, spanning many iterations of the site.

In order to improve the codebase, Etsy made the decision to adopt TypeScript, a superset of JavaScript with the optional addition of types. This means that any valid JavaScript code is valid TypeScript code, but TypeScript provides additional features on top of JS (the type system).

Based on [research at Microsoft](#), static type systems can heavily reduce the amount of bugs in a codebase. Microsoft researchers found that using TypeScript or [Flow](#) could have prevented 15% of the public bugs for JavaScript projects on Github.

Strategies for Adoption

There are countless different strategies for migrating to TypeScript.

For example, Airbnb [automated as much of their migration as possible](#) while other companies enable less-strict TypeScript across their projects, and add types to their code over time.

In order to determine their strategy, Etsy had to answer a few questions...

1. How strict do they want their flavor of TypeScript to be? - TypeScript can be [more or less “strict”](#) about checking the types in your codebase. A stricter configuration results in stronger guarantees of program correctness.
TypeScript is a superset of JavaScript, so if you wanted you could just rename all your .js files to .ts and still have valid TypeScript, but you would not get strong guarantees of program correctness.

2. How much of their codebase do they want to migrate? - TypeScript is designed to be easily adopted *incrementally* in existing JavaScript projects. Again, TypeScript is a superset of JavaScript, so all JavaScript code is valid TypeScript. Many companies opt to gradually incorporate TypeScript to help developers ramp up.
3. How specific do they want the types they write to be? - How accurately should a type fit the thing it's describing? For example, let's say you have a function that takes in the name of an HTML tag. Should the parameter's type be a string? Or, should you create a map of all the HTML tags and the parameter should be a key in that map (far more specific)?

```
// This function type-checks, but I could pass in literally any string in
// as an argument.
function makeElement(tagName: string): HTMLElement {
  return document.createElement(tagName);
}

// This throws a DOMException at runtime
makeElement("literally anything at all");
```

1.

```
// This function makes sure that I pass in a valid HTML tag name as an
// argument.
// It makes sure that 'tagName' is one of the keys in
// HTMLElementTagNameMap, a built-in type where the keys are tag names
// and the values are the types of elements.
function makeElement(tagName: keyof HTMLElementTagNameMap): HTMLElement {
  return document.createElement(tagName);
}

// This is now a type error.
makeElement("literally anything at all");

// But this isn't. Excellent!
makeElement("canvas");
```

Based on the previous questions, Etsy's adoption strategy looked like

1. Make TypeScript as strict as reasonably possible, and migrate the codebase file-by-file.
2. Add really good types and really good supporting documentation to all of the utilities, components, and tools that product developers use regularly.
3. Spend time teaching engineers about TypeScript, and enable TypeScript syntax team by team.

To elaborate more on each of these points...

Gradually Migrate to Strict TypeScript

Etsy wanted to set the compiler parameters for TypeScript to be as strict as possible.

The downside with this is that they would need *a lot* of type annotations.

They decided to approach the migration incrementally, and first focus on typing actively-developed areas of the site.

Files that had reliable types were given the .ts file extension while files that didn't kept the .js file extension.

Make sure Utilities and Tools have good TypeScript support

Before engineers started writing TypeScript, Etsy made sure that all of their tooling supported the language and that all of their core libraries had usable, well-defined types.

In terms of tooling, Etsy uses Babel and the plugin [babel-preset-typescript](#) that turns TypeScript into JavaScript. This allowed Etsy to continue to use their existing build infrastructure. To check types, they run the TypeScript compiler as part of their test suite.

Etsy makes heavy use of custom ESLint linting rules to maintain code quality.

They used the [TypeScript ESLint](#) project to get a handful of TypeScript specific linting rules.

Educate and Onboard Engineers Team by Team

The *biggest* hurdle to adopting TypeScript was getting everyone to learn TypeScript.

TypeScript works better the more types there are. If engineers aren't comfortable writing TypeScript code, fully adopting the language becomes an uphill battle.

Etsy has several hundred engineers, and very few of them had TypeScript experience before the migration.

The strategy Etsy used was to onboard teams to TypeScript gradually on a team by team basis.

This had several benefits

- Etsy could refine their tooling and educational materials over time. Etsy found a course from [ExecuteProgram](#) that was great for teaching the basics of TypeScript in an interactive and effective way. All members of a team would have to complete that course before they onboarded.
- No engineer could write TypeScript without their teammates being able to review their code. Individual engineers weren't allowed to write TypeScript code before the rest of their team was ready.
- Engineers had plenty of time to learn TypeScript and factor it into their roadmaps. Teams that were about to start new projects with flexible deadlines were the first to onboard TypeScript.

Managing Infrastructure with Code at Shopify

Shopify is a tech company that helps businesses build e-commerce stores. If you want to build an e-commerce store to sell your handcrafted guitars, you could use Shopify to set up your website, manage customer information, handle payments/banking and more.

Jeremy Cobb is a software engineer at Shopify, where he works on the Contact Center team. They're responsible for building the tooling that helps Shopify's customer service team deal with all the support inquiries from businesses that use the platform.

He wrote a great [blog post](#) on how his team uses [Terraform](#) for configuration management. Terraform is an open source tool that lets you configure your infrastructure using code.

Here's a summary

The Contact Center team builds the tooling that Shopify customer service agents use to handle support requests.

One tool the engineers rely on is [Twilio's](#) TaskRouter service. Twilio is a company that builds programmable communication tools, so you can use Twilio's API for sending emails, text messages, etc.

Shopify uses Twilio TaskRouter to handle routing communication tasks (voice, chat, etc.) to the most appropriate customer service agent based on a set of routing rules. For example, users in the US might get sent to a different customer service agent than users in Canada.

Previously, Shopify would configure these routing rules using Twilio's website. However, the complexity of the rules grew and it became too much for a single person to manage.

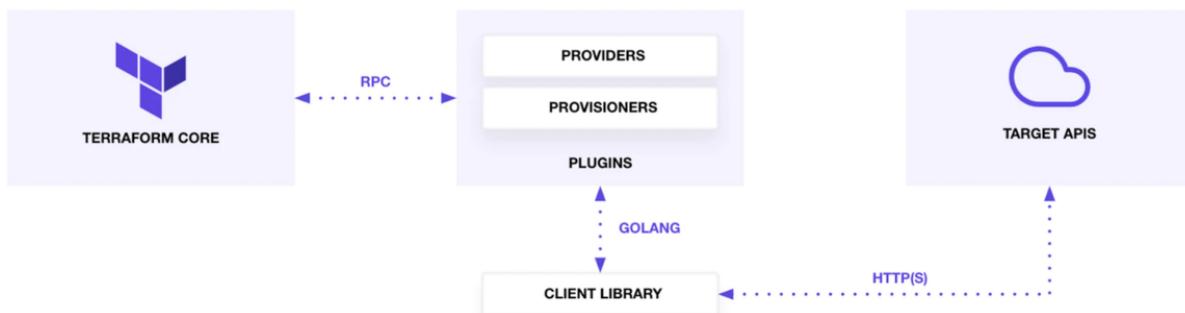
Having multiple people manage the rules quickly became troublesome because the website doesn't provide a clear history of changes or way to roll changes back.

In order to solve this, the Contact Center team decided to use Terraform to manage the configuration of Twilio Taskrouter.

Terraform is an open source tool that lets you write code to manage/configure your infrastructure/tooling. You can write the code in JSON or in a Terraform-specific language called HashiCorp Configuration Language (HCL).

In order to use Terraform to manage your infrastructure, you need 3 things.

1. A reliable API - The infrastructure/service (Twilio in Shopify's case) will need a reliable API that you can send requests to in order to make changes. If the only way of configuring your infrastructure is through their website, then it's not possible to use any infrastructure as code solutions.
2. A Terraform Provider - In order to consume the infrastructure's API, Terraform needs a Provider Plugin, which lets Terraform interface with external APIs. The Provider Plugin contains CRUD instructions for all the resources that the Provider manages. For example, the AWS Terraform Provider Plugin will have CRUD instructions for AWS ec2 resources. All the major cloud computing companies (GCP, AWS, etc.) maintain their own Terraform Providers for their service. You can also create your own Providers to use external APIs that don't already have a provider available.
3. A Client Library - You'll also want a separate library that the Terraform Provider can interface with to make API requests to the external infrastructure API. You *could* create a Terraform Provider Plugin that makes the API calls itself, but this is highly discouraged. It's better to modularize the API calls in a separate client library.



So, Twilio TaskRouter provided a reliable API that the Shopify team could use to manage their rule configuration.

There was no TaskRouter Terraform Provider available at the time (Twilio has since developed [their own](#)) so the Shopify team built one themselves.

The Provider defines how Terraform should manage Twilio TaskRouter. It contains resource files for every type of resource in TaskRouter that Terraform has to manage; each resource file has CRUD instructions that tell Terraform how to manage it.

The Provider also has import instructions that let Terraform import existing infrastructure. This is useful if you already have infrastructure running and want to start using Terraform to manage it.

The Shopify team also built a client library that the Terraform Provider would use to make HTTP calls to Twilio's API.

Using Terraform

With Terraform set up, Shopify could stop relying on Twilio's website for configuring TaskRouter rules and instead write them using HCL (Terraform's domain specific language).

This made seeing changes to the infrastructure much easier and allowed Shopify to integrate software engineering practices like pull requests, code reviews, etc for their TaskRouter rules.

It also allowed non-developers to start configuring rule changes themselves. Business and support teams could write rule changes in HCL and create PRs instead of making a request and waiting for a developer to log onto Twilio's website and change the config manually.

For more details on how Shopify created the Provider and on how they use Terraform, you can read the full article [here](#).

Sharding Databases at Quora

Quora is a social platform where users can post and answer questions on anything. The website receives more than 600 million visits per month.

Quora relies on MySQL to store critical data like questions, answers, upvotes, comments, etc. The size of the data is on the order of tens of terabytes (without counting replicas) and the database gets hundreds of thousands of queries per second.

Vamsi Ponnekanti is a software engineer at Quora, and he wrote a great [blog post](#) about why Quora decided to shard their MySQL database.

MySQL at Quora

Over the years, Quora's MySQL usage has grown in the number of tables, size of each table, read queries per second, write queries per second, etc.

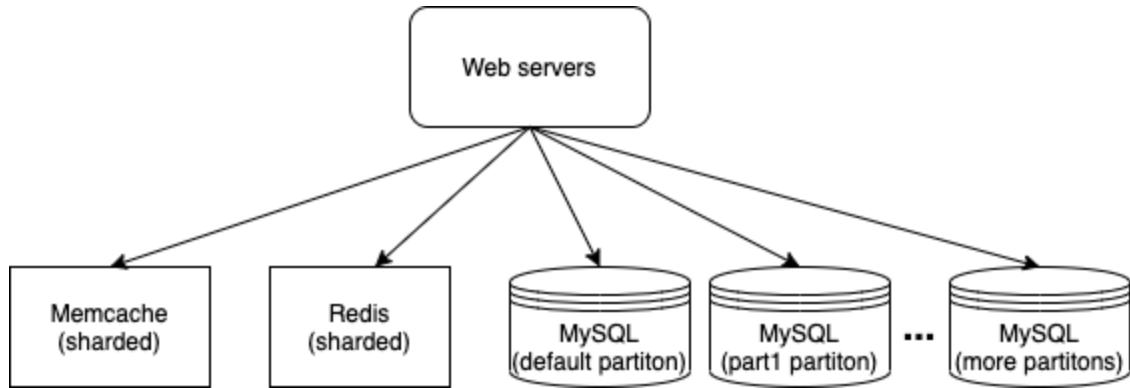
In order to handle the increase in read QPS (queries per second), Quora implemented caching using Memcache and Redis.

However, the growth of write QPS and growth of the size of the data made it necessary to shard their MySQL database.

At first, Quora engineers split the database up by tables and moved tables to different machines in their database cluster.

Afterwards, individual tables grew too large and they had to split up each logical table into multiple physical tables and put the physical tables on different machines.

We'll talk about how they implemented both strategies.



Splitting by Table

As the read/write query load grew, engineers had to scale the database horizontally (add more machines).

They did this by splitting up the database tables into different partitions. If a certain table was getting very large or had lots of traffic, they create a new partition for that table. Each partition consists of a master node and replica nodes.

The mapping from a partition to the list of tables in that partition is stored in ZooKeeper.

The process for creating a new partition is

1. Use mysqldump (a tool to generate a backup of a MySQL database) to dump the table in a single transaction along with the current binary log position (the binary log or binlog is a set of log files that contains all the data modifications made to the database)
2. Restore the dump on the new partition
3. Replay binary logs from the position noted to the present. This will transfer over any writes that happened after the initial dump during the restore process (step 2).
4. When the replay is almost caught up, the database will cutover to the new partition and direct queries to it. Also, the location of the table will be set to the new partition in ZooKeeper.

A pro of this approach is that it's very easy to undo if anything goes wrong. Engineers can just switch the table location in ZooKeeper back to the original partition.

Some shortcomings of this approach are

- Replication lag - For large tables, there can be some lag where the replica nodes aren't fully updated.
- No joins - If two tables need to be joined then they need to live in the same partition. Therefore, joins were strongly discouraged in the Quora codebase so that engineers could have more freedom in choosing which tables to move to a new partition.

Splitting Individual Tables

Splitting large/high-traffic tables onto new partitions worked well, but there were still issues around tables that became very large (even if they were on their own partition).

Schema changes became very difficult with large tables as they needed a huge amount of space and took several hours (they would also have to frequently be aborted due to load spikes).

There were unknown risks involved as few companies have individual tables as large as what Quora was operating with.

MySQL would sometimes choose the wrong index when reading or writing. Choosing the wrong index on a 1 terabyte table is much more expensive than choosing the wrong index on a 100 gigabyte table.

Therefore, engineers at Quora looked into sharding strategies, where large tables could be split up into smaller tables and then put on new partitions.

Key Decisions around Sharding

When implementing sharding, engineers at Quora had to make quite a few decisions. We'll go through a couple of the interesting ones here. Read the full article for more.

Build vs. Buy

Quora decided to build an in-house solution rather than use a third-party MySQL sharding solution ([Vitess](#) for example).

They only had to shard 10 tables, so they felt implementing their own solution would be faster than having to develop expertise in the third party solution.

Also, they could reuse a lot of their infrastructure from splitting by table.

Range-based sharding vs. Hash-based sharding

There are different [partitioning criteria](#) you can use for splitting up the rows in your database table.

You can do range-based sharding, where you split up the table rows based on whether the partition key is in a certain range. For example, if your partition key is a 5 digit zip code, then all the rows with a partition key between 7000 and 79999 can go into one shard and so on.

You can also do hash-based sharding, where you apply a hash function to an attribute of the row. Then, you use the hash function's output to determine which shard the row goes to.

Quora makes frequent use of range queries so they decided to use range-based sharding. Hash-based sharding performs poorly for range queries.

How Quora Shards Tables

So, when Quora has a table that is extremely large, they'll split it up into smaller tables and create new partitions that hold each of the smaller tables.

Here are the steps they follow for doing this

1. Data copy phase - Read from the original table and copy to all the shards.
Quora engineers set up N threads for the N shards and each thread copies data to one shard. Also, they take note of the current binary log position.
2. Binary log replay phase - Once the initial data copy is done, they replay the binary log from the position noted in step 1. This copies over all the writes that happened during the data copy phase that were missed.
3. Dark read testing phase - They send shadow read traffic to the sharded table in order to compare the results with the original table.
4. Dark write testing phase - They start doing dark writes on the sharded table for testing. Database writes will go to both the unsharded table and the sharded table and engineers will compare.

If Quora engineers are satisfied with the results from the dark traffic testing, they'll restart the process from step 1 with a fresh copy of the data. They do this because the data may have diverged between the sharded and unsharded tables during the dark write testing.

They will repeat all the steps from the process until step 3, the dark read testing phase. They'll do a short dark read testing as a sanity check.

Then, they'll proceed to the cutover phase where they update ZooKeeper to indicate that the sharded table is the source of truth. The sharded table will now serve read/write traffic.

However, Quora engineers will still propagate all changes back to the original, unsharded table. This is done just in case they need to switch back to the old table.

This article was published in 2020 and Quora had successfully sharded 3 large production tables before the article was written.

For more details, you can read the full article [here](#).

Video Delivery at Twitter with HTTP Live Streaming

Tinder is one of the largest dating apps in the world with more than 75 million monthly active users.

One of their product features is *Swipe Night*, which is a choose-your-own-adventure game built in the app. You watch short video clips (themed around a solve-the-mystery game) and you make choices on what the main character should do. Tinder will match you with possible dates based on your choices.

Everyone on the app is watching the same video clips and it's done live - at 6 p.m. local time.

Shreyas Hirday is a senior software engineer at Tinder and he wrote a great [blog post](#) on the technology Tinder used to stream the video clips to millions of users simultaneously.

Here's a summary

There are many ways to deliver video content. The best approach depends on the tradeoffs you're making.

In Tinder's case, they cared about

- Dynamic - Tinder should have the ability to change the video content at any time.
- Efficient - Memory usage should be minimized since Tinder is being run on mobile devices.
- Seamless - There should be little to no interruption during playback.

Based on these goals, Tinder decided to use [HTTP Live Streaming \(HLS\)](#), an [adaptive bitrate protocol](#) developed by Apple.

Adaptive bitrate streaming just means that the server will have different versions of the video and each version differs in display size (resolution) and file size (bitrate).

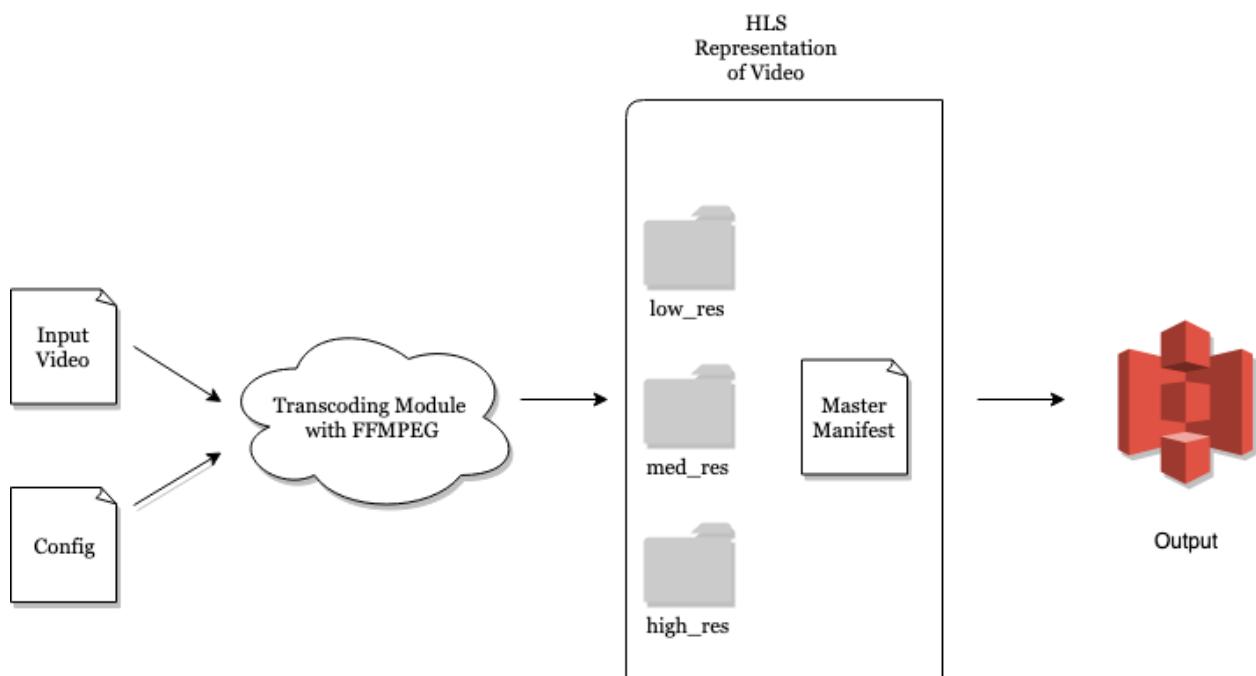
The video player will dynamically choose the best video version based on the user's screen size and bandwidth. It will choose the version that minimizes buffering and gives the best user experience.

An HLS stream will provide a manifest file to the video player, which includes the URL to each copy of the video as well as the level of bandwidth the user should have in order to view that level of quality without issue.

Transcoding

Tinder engineers used [FFMPEG](#) to transcode MP4 files to HLS streams. They developed a workflow that had the MP4 file and configurations (resolution, bitrate, frame rate, etc.) as input and a directory containing the HLS stream as output.

They had multiple configurations for all the different video versions they wanted and they stored all these video versions in an AWS S3 bucket.



Some of the different configuration options they had for the various versions of the videos were

- Frame rate
- Video Resolution
- Desired Video & Audio Bitrate
- Video Bitrate Variability
- Segment Length
- Optimizations

You can read the article for a discussion on how they configured each of these parameters.

Validation

The output directory will have a Master Manifest file with information about all the different video versions in the HLS stream.

The video player will then decide which version to play and whether it should switch to a lower file-size version based on the information in the manifest file.

Therefore, having an accurate manifest file is very important for the user experience.

Apple provides a [Media Stream Validator](#) tool that tests the manifest by simulating a streaming experience. Tinder uses the results from that test to update the manifest and ensure accuracy.

Tinder then places the finalized manifest and videos in their production AWS S3 bucket.

Content Access & Delivery

Tinder uses AWS Cloudfront, a content delivery network (CDN), to ensure low-latency streaming for all their users.

As users from different areas of the US start playing SwipeNight, the CDN will copy the HLS stream directory from AWS S3 into regional caches so users from that region can get low latency access.

Measuring Video Performance

Tinder uses 5 key performance indicators (KPIs) to measure how well the streaming works

1. Perceived start up time for a user
2. Number of stalls during playback
3. Time spent in the stalled state
4. % of video sessions with errors
5. Average quality of the video measured by average bitrate

The Tinder app measures these KPIs along with metadata about the device and its network connection.

Tinder then works to find the right balance between these 5 KPIs for their use case. For a traditional streaming app like Netflix, spending 5 seconds in a buffering state might not be that bad. But a 5 second buffer on a mobile-centric app like Tinder can feel like an eternity.

For more details, you can read the full blog post [here](#).

Site Reliability Engineering at BlackRock

BlackRock is the world's biggest Asset Manager with more than \$10 trillion in assets under management.

In addition to being an asset manager, BlackRock is also a technology company. They sell a variety of software to other asset managers, banks, insurance companies, etc.

Their biggest product is Aladdin, the financial industry's most popular software platform for investment management. Asset managers (banks, pension funds, hedge funds, etc.) use Aladdin to track profit/loss, manage portfolio risk, make trades, analyze historical data, etc.

In 2013, the Aladdin platform was used to manage more than 7% of the world's 225 trillion dollars of financial assets (and it's grown since then), so any issues with the platform can have major consequences on the global financial system.

BlackRock's Site Reliability Engineering team has built a robust telemetry platform to oversee the health, performance and reliability of Aladdin.

Sudipan Mishra is an engineer on BlackRock's SRE team and he wrote a great [blog post](#) on the architecture of their Telemetry platform.

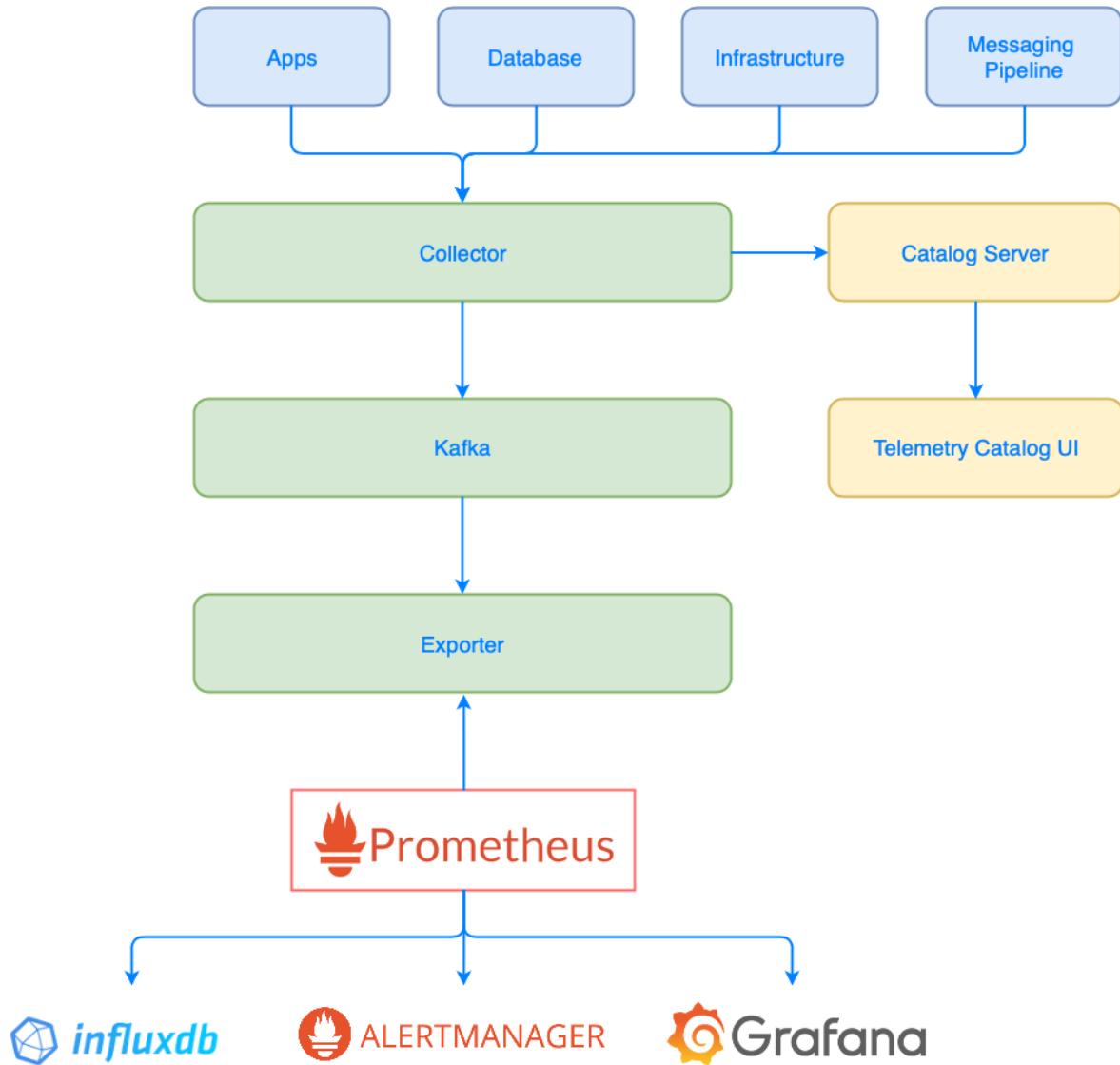
Here's a summary

Architecture of the Telemetry Platform

All the various components of Aladdin generate large amounts of logs, data, etc.

The Telemetry platform is responsible for aggregating all these reports, displaying them, and sending alerts to the various Aladdin developers at BlackRock if one of their services is not performant.

Here's the architecture.



At the top you have all the various apps, databases, infrastructure and pipelines involved in the Aladdin platform. All of these report metrics which are then read by the Telemetry platform's collectors.

From the collector, these metrics go to a Catalog server, an internally developed service that manages which metrics should be catalogued.

Some metrics might be too noisy/unnecessary so engineers can remove them from the Telemetry Catalog UI.

The metrics that get passed on are then sent from the Collector to Kafka and eventually to [Prometheus](#). Prometheus is an open source systems monitoring and alerting toolkit that was originally developed at SoundCloud.

From Prometheus, the metrics go to InfluxDB, AlertManager and Grafana.

InfluxDB is an open source time series database that BlackRock uses for long-term storage of all the telemetry metrics.

AlertManager is a tool in the Prometheus toolkit that triggers alerts to BlackRock engineers based on the metrics.

Grafana is another Prometheus tool that lets engineers produce dashboards and charts to visualize the telemetry metrics.

Alerting Strategy

Prometheus' AlertManager will send alerts to the various developers working on Aladdin based on the telemetry. A team should be alerted as quickly as possible if any incidents have an impact on their service.

However, if the SRE team isn't careful about how they implement alerts then they can cause things like [alarm fatigue](#).

The SRE team has 4 core principles for their alerts

1. Actionable - Every alert should clearly define what is broken or about to break. Alerts should also propose the corrective actions to take.
2. Effective - False positives (issuing an alert when there is no incident) and False negatives (not triggering an alert despite there being an issue) must both be minimized/eliminated. Otherwise, they can cause mistrust in the alerting system.
3. Impactful - Developers should not be getting alerts for trivial/unimportant things. Otherwise, developers can get alarm fatigue and accidentally ignore important alerts.

4. Transparent - As developers onboard new applications, they should know what alerts they're going to be getting. They should also have an idea of how many alerts they'll typically see for that app.

BlackRock evaluated different types of alert systems to meet all of these principles.

You can read about some of the options in [Google's SRE book](#). The book is free and definitely a must-read if you're interested in SRE.

BlackRock's SRE team decided to go with a *Multiwindow, Multi-Burn-Rate Alerting Strategy*. This is #6 on Google's list of alert types in the Google SRE Workbook.

You see how much error is allowable and set various limits around that. As you notice errors in the telemetry, you "burn" against the allowable error limit. Once you surpass that allowable limit (and if there are still errors coming) then you send an alert.

BlackRock found this strategy to have a low false-positive rate, a low false-negative rate, a fast detection time and a very low reset time.

In order to test their alerting strategy, they wrote a script that would let developers extract metrics from their Prometheus instance for a given time and date range. They can take those metrics and then backtest their alert strategy to see how many alerts they would've gotten and whether there would've been any false negatives or false positives.

For more details, you can read the full article [here](#).

How Clubhouse Recommends Rooms

Clubhouse is a social audio application that allows you to create private audio rooms where you can speak to your friends in the app. You can also create public audio rooms where other users of the app can join in the audience (and listen in) or as a speaker.

The app experienced viral growth in 2020, peaking in February 2021 with Elon Musk interviewing Vlad Tenev (the CEO of Robinhood) on the whole GameStop saga. Since then, however, usage of the Clubhouse app has plummeted and downloads of the app have been stagnating.

One of the reasons why is because Clubhouse's room recommendations (the audio rooms recommended when you open the app) were pretty poor.

Speaking from my personal usage, the room recommendations were not relevant to my interests so I'd frequently open the app, find nothing interesting and immediately leave.

Akshaan Kakar is a software engineer working on machine learning & discovery at Clubhouse, and he wrote a great [blog post](#) on how Clubhouse built a new recommendation engine to provide better room recommendations.

Here's a summary

When you first open the Clubhouse app, you are in the “hallway”. In the hallway are a bunch of different audio rooms that you can join.



At any given time, there are thousands of possible rooms that Clubhouse can recommend to you in your hallway. Therefore they need to rank the potential rooms and present you with the top recommendations.

Early on, Clubhouse used simple heuristics to rank the rooms. Heuristics like how many of your friends were in the room or how closely the room matched with the topics that you're following.

Now, Clubhouse uses machine learning with a ranking model that is based on Gradient Boosted Decision Trees (GBDTs).

If you're unfamiliar with GBDTs, [this](#) is the best article I found that explains them. It starts from decision trees, goes into ensemble learning (a model that makes predictions by combining multiple simpler models) and boosting and then goes into GBDTs.

The GBDT model at Clubhouse is based on hundreds of different data points like whether you spend more time in smaller rooms vs. larger rooms, whether you prefer to speak or just listen in, how many participants are in the room, etc.

The model is trained as a [classifier](#), where it will create a score for each room between 0 and 1, where 0 means the room is not relevant to you at all while 1 means it's extremely relevant.

This classification score is then used to rank the rooms in your hallway.

Complexities

There are many complexities that arise with the machine learning model. Here are a couple.

Real Time Features

Many of the [features](#) (data points) used by the ranking model are slow-moving batch features that can be computed once every few hours.

However, the model is recommending live rooms that change on a second by second basis. A celebrity can randomly join a room and that completely changes how the room should be ranked.

Therefore, engineers also have to incorporate real time data into the recommendation model.

To do this, they have individual events that fire every time there is a change to a room. These events are then sent as streaming data to Clubhouse's recommendation model so it can incorporate real time information into its recommendations.

These real time features are also logged at inference time, so engineers can use the values later to train future iterations of the model.

Making Fast Inferences

Clubhouse has to run this recommendation model on hundreds of features across hundreds of rooms. This can be very resource-intensive so they've taken steps to make sure user experience isn't compromised.

They use a simple memory-backed feature storage mechanism, so fetching model features is done quickly.

They also spin up lightweight stateless microservices that are solely responsible for model inference. The server will fetch the feature data and then send it to the microservice responsible for machine-learning inferences. With this set up, model-inference is isolated from the core server and it can be scaled up/down independently.

For more details, read the full article [here](#).

Continuous Delivery at Airbnb

Airbnb has recently [migrated](#) from a Ruby on Rails monolith to a Services-Oriented Architecture (SOA).

This migration helped Airbnb scale their application, but it also introduced new challenges.

One of these challenges was around Airbnb's Continuous Delivery process and how they had to adapt it to the new services architecture.

Jens Vanderhaeghe is a senior software engineer at Airbnb and Manish Maheshwari is a lead product manager. They wrote a great [blog post](#) on Airbnb's new continuous delivery process and how they migrated.

Here's a summary

Previously, Airbnb used an internal service called [Deployboard](#) to handle their deploys. Deployboard worked great when Airbnb was using a Ruby on Rails monolith but over the past few years the company has shifted to a Microservices-oriented Architecture.

A microservices architecture means decentralized deployments where individual teams have their own pipeline.

Airbnb needed something more templated, so that each team could quickly get a standard, best-practices pipeline, rather than building their own service from scratch.

[Spinnaker](#) is an open source continuous delivery platform that was developed internally at Netflix and further extended by Google.

Airbnb decided to adopt Spinnaker because it

- was shown to work at Airbnb's scale by Google and Netflix
- allows you to easily plug in custom logic so you can add/change functionality without forking the core codebase

- automates Canary analysis. Canary deployments let you expose the new version of the app to a small portion of your production traffic and analyze the behavior for any errors. Spinnaker helps automate this.

Migrating to Spinnaker

Airbnb has a globally distributed team of thousands of software engineers. Getting all of them to shift to Spinnaker would be a challenge.

They were particularly worried about the *Long-tail Migration Problem*, where they could get 80% of teams to switch over to the new deployment system but then struggle to get the remaining 20% to switch over.

Being forced to maintain two deployment systems can become very costly and is a reliability/security risk because the legacy system gets less and less maintenance/attention over time.

To prevent this, Airbnb had a migration strategy that focused on 3 pillars.

1. Focus on Benefits
2. Automated Onboarding
3. Provide Data

Focus on Benefits

Airbnb started by encouraging teams to adopt Spinnaker voluntarily.

They did this by first onboarding a small group of early adopters. They identified a set of services that were prone to causing incidents and switched those teams over to Spinnaker.

The automated Canary analysis quickly demonstrated its value to those teams as well as the other features that Spinnaker provided.

These early adopters ended up becoming evangelists for Spinnaker and spread the word to other teams at Airbnb organically. This helped increase voluntary adoption.

Automated Onboarding

As more teams started adopting Spinnaker, the Continuous Delivery team at Airbnb could no longer keep up with demand. Therefore, they started building tooling to automate the onboarding process to Spinnaker.

They created an abstraction layer on top of Spinnaker that let engineers make changes to the CD platform with code ([IaC](#)). This allowed all continuous delivery configuration to be source controlled and managed by Airbnb's tools and processes.

Data

The Continuous Delivery team also put a great amount of effort into clearly communicating the value-add of adopting Spinnaker.

They created dashboards for every service that adopted Spinnaker to show metrics like number of regressions prevented, increase in deploy frequency, etc.

Final Hurdle

With this 3 pillar strategy, the vast majority of teams at Airbnb had organically switched over to Spinnaker.

However, adoption began to tail off as the company reached ~85% of deployments on Spinnaker.

At this point, the team decided to switch strategy to avoid the long-tail migration problem described above.

Their new plan consisted of

1. Stop the bleeding - Stop any new services/teams from being deployed using the old continuous delivery platform.
2. Announce deprecation date - Announce a deprecation date for the old continuous delivery platform and add a warning banner at the top.
3. Send out automated PRs - Airbnb has an in-house refactor tool called Refactorator that helped with making the switch to Spinnaker easier.

4. Deprecate and post-deprecation - On deprecation date, they had code in-place that blocked deploys from the old continuous delivery platform. However, they had exemptions in-place for emergencies where the old system had to be used.

Conclusion

With this strategy, Airbnb was able to get to the 100% finish line in the migration.

This migration serves as the blueprint for how other infrastructure-related migrations will be done at Airbnb.

Read the [full article](#) for more details.

How BuzzFeed optimized their Frontend

In June of 2021, Google made a big change to their search algorithm with the [Page Experience Update](#).

With this update, Google would start using their Core Web Vital metrics as a factor in their page rankings. Sites with poor Core Web Vitals would rank lower in Google search results.

Core Web Vitals are a set of standardized metrics that can measure how good of a user experience a website is giving. They currently focus on 3 metrics

1. Largest Contentful Paint (LCP) - how many seconds does a website take to show the user the largest [content](#) (text or image block) on the screen? A good LCP score would be under 2.5 seconds.
2. First Input Delay (FID) - how much time does it take from when a user first interacts with the website (click a link, tap a button, etc.) to the time when the browser is able to respond to that interaction? A good FID score is under 100 milliseconds.
3. Cumulative Layout Shift (CLS) - How much does a website unexpectedly shift during its lifespan? A large paywall popping up 10 seconds after the content loads is an example of an unexpected layout shift that will cause a negative user experience. You can read about how the CLS score is calculated [here](#). Google has come up with two metrics: impact fraction and distance fraction, and they multiply those two to calculate the CLS score.

BuzzFeed is a digital media company that covers content around pop culture, movies, tv shows, etc. and they get a significant part of their traffic from Google Search (more than 100 million visits per month). Having their articles rank high on the google search results page is extremely critical to their business.

Edgar Sanchez is a software engineer at BuzzFeed, and he wrote a great 3 part [series](#) on how BuzzFeed fixed their Core Web Vitals to meet Google's standards. More specifically,

how BuzzFeed fixed their CLS score (their LCP and FID scores had already met Google's standards).

Here's a Summary

When Google announced that they'd be factoring Core Web Vitals into PageRank, engineers at BuzzFeed took note.

They checked their Largest Contentful Paint (LCP), First Input Delay (FID) and Cumulative Layout Shift (CLS) scores.

Their LCP and FID scores were fine. However, their CLS score was very poor.

Only 20% of visits to BuzzFeed were achieving a “good” experience (a [CLS score](#) of less than 0.1). In order to pass Google's Core Vitals [test](#), 75% of visits should get a CLS score of less than 0.1.

The first step in addressing this issue was to improve Observability over CLS, so engineers could figure out the cause of the issue.

To increase Observability, BuzzFeed used two tools.

1. Synthetic Monitoring - Use [Calibre](#) to create a testing environment and run CLS tests several times a day.
2. Real User Monitoring - Add analytics metrics to the frontend that measure how much CLS users are experiencing. Hence monitor *real users*.

BuzzFeed started with Synthetic Monitoring.

They broke their web pages down into independently testable layers to help make tests more consistent.

- Content Layer - just the page content. So, the article, any quizzes, interactive embeds etc.
- Feature Layer - Include everything above (page content) but also include complimentary units like a comment section, polls, trending feed, etc.

- Full Render Layer - Include everything above (features + content) but also include ads.

They loaded a couple hundred pages into Calibre and ran tests to figure out what was causing the CLS issues.

With Synthetic Monitoring, they were quickly able to narrow in on some of the causes for the issues.

They took a data-driven approach to prioritizing optimizations, and looked at which page types/units achieved the highest volume of page views. Engineers optimized CLS on those pages first.

However, even after solving the biggest issues that were apparent from their tests in Calibre, BuzzFeed was still unable to get their CLS score above Google's threshold.

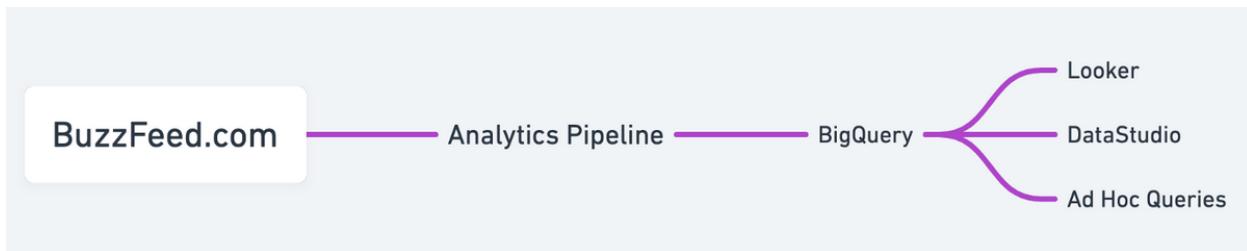
Therefore, they turned to Real User Monitoring (RUM).

With RUM, BuzzFeed would lean on their massive audience (more than 100 million visits per month), their analytics pipeline and the [Layout Instability API](#).

The Layout Instability API provides 2 interfaces for measuring and reporting layout shifts so you can send that data to your backend server.

BuzzFeed has an in-house analytics pipeline that they use for keeping track of various types of real user monitoring data, so they hooked the pipeline up with the Layout Instability API.

The data travels from the frontend through various filters before being stored in [BigQuery](#) (data warehouse from Google). From there, engineers can run analyses or export the data to analysis tools like [Looker](#) and [DataStudio](#).



Engineers looked at page volume and CLS scores to figure out which areas to target and where they should spend their time optimizing.

Optimizations

Here are some of the common issues BuzzFeed solved that resulted in improvements to their CLS scores

- Correct Image Sizing - All images should have width/height attributes.
- Static Placeholder for Ads - BuzzFeed has ads that will change dimensions depending on which ad is being served. They looked at the most common ad sizes and created static placeholders for them so the page wouldn't change suddenly once an ad was loaded.
- Static Placeholders for Embedded Content - BuzzFeed frequently embeds content from other websites (Tweets, TikTok video, YouTube, etc.). However, finding the dimensions for static placeholders for embedded content was quite difficult due to the huge variety of content sizes.

The most difficult to solve was generating static placeholders for embedded content since many embeds have no fixed dimensions and are difficult to accurately size. Embedding a tweet, for example, can vary dramatically in height depending on the content of the tweet and whether it contains an image/video.

BuzzFeed engineers solved this by gathering embed dimensions from all their pages and collecting them in their analytics pipeline and eventually in BigQuery.

Now, when a page is requested, the rendering layer will check BigQuery for the dimensions of the embedded content and add correctly-sized placeholders for the content.

As new pages get published, the dimensions of any third party embeds on those pages will be loaded into BigQuery.

Result

With these changes, BuzzFeed was able to achieve ~80% of all page views having a good CLS score. This is a massive improvement from their starting point of ~20% of page views.

For more details, you can read the full series [here](#).

The Evolution of Benchling's Search Architecture

Benchling is a cloud platform for biotech research & development. Scientists can use the platform to keep track of their experiments, collaborate with other teams, and analyze experiment data.

Over 200,000 researchers use Benchling as a core part of their workflow when running experiments.

Matt Fox is a software engineer at Benchling and he wrote a great [blog post](#) on the architecture of their Search System.

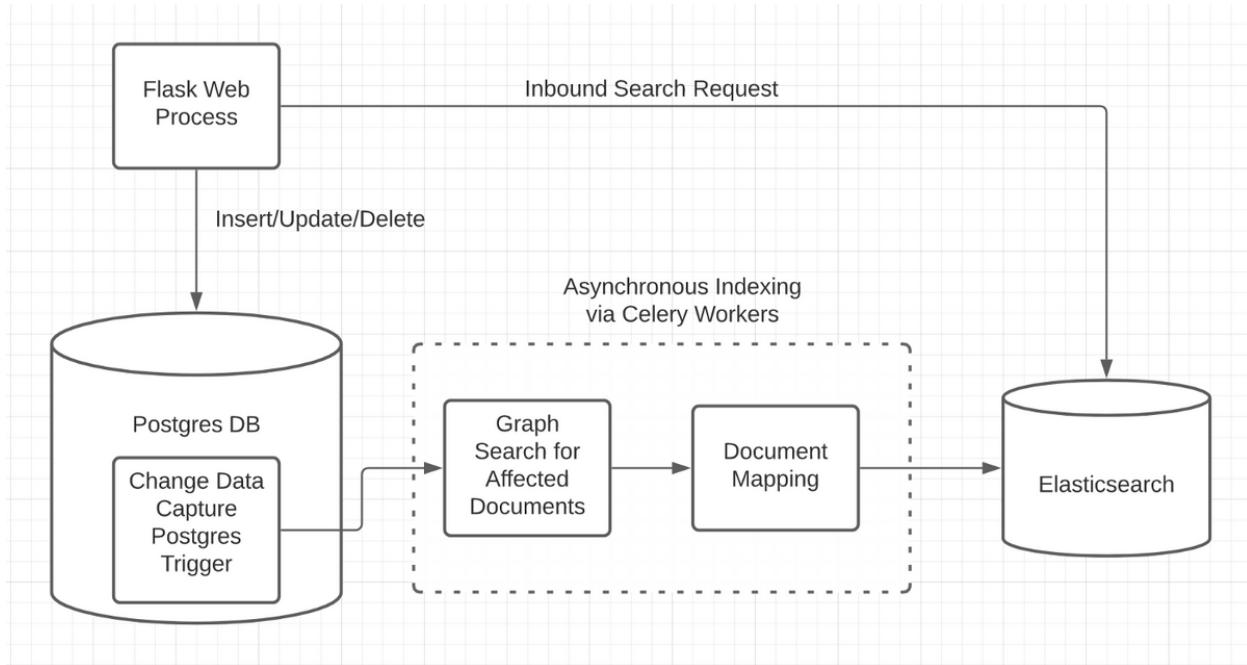
Here's a summary

Benchling's search feature is a core part of the platform. Researchers can use the search feature to find whatever data they have stored; whether it's specific experiments, DNA sequences, documents, etc.

The search system also has full-text search capabilities so you can search for certain keywords across all the contents that you've stored on Benchling.

Benchling's Initial Search Architecture (2015 - 2019)

The initial architecture is described in the picture above.



The user would interact with the system through a web server written in Flask.

If the user wanted to perform a [CRUD](#) action like creating an experiment or deleting a project, then that would be carried out using the core Postgres database.

If the user wanted to search for something, then that would be done by sending a search request to an Elasticsearch cluster that was kept synced with the Postgres database.

Benchling managed the syncing with a data pipeline that copied any CRUD updates to Postgres over to Elasticsearch.

Whenever the user created/read/updated/deleted something in Postgres, then

1. [Postgres triggers](#) would trigger if a searchable item was changed (items that were not searchable did not need to be stored in Elasticsearch). They would send the changes to a Task Queue ([Celery](#)).
2. [Celery Workers](#) would determine all the different documents in Elasticsearch that needed to be updated as a result of the CRUD action. There could be multiple documents that needed to be updated because the data was [denormalized](#) when stored in Elasticsearch (we'll talk about why below).

3. All the necessary updates would be pushed to Elasticsearch.

This architecture worked well, but had several pain points with the main one being keeping Elasticsearch synced with Postgres. There was too much replication lag.

With the Elasticsearch cluster, fast reads were prioritized (so users could get search results quickly) and data was denormalized when transferred from Postgres.

Denormalization is where you write the same data multiple times in the different documents instead of using a relation between those documents. This way, you can avoid costly joins during reads. Data denormalization improves read performance at the cost of write performance.

The increased cost in write performance is caused by write amplification, where a change to one row in Postgres expands to many document updates in Elasticsearch since you have to individually update *all* the documents that contain that value.

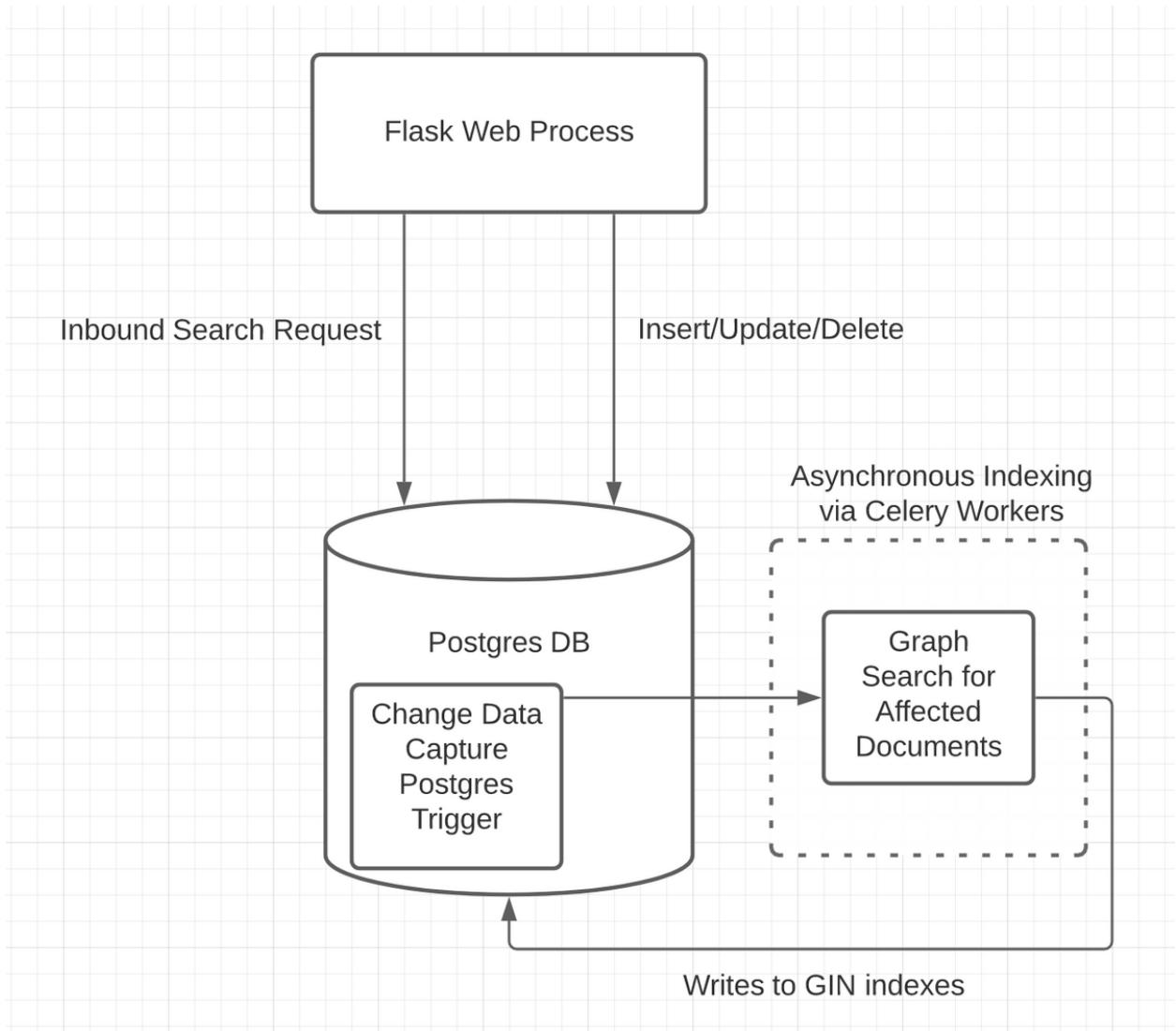
These costlier writes meant more lag between updating the Postgres database and seeing that update reflected in Elasticsearch. This could be confusing to users as someone might create a new project but then not see it appear if he searches the project name immediately after.

The Benchling team was also dealing with a lot of overhead and complexity with maintaining this search system. They had to maintain *both* Postgres and Elasticsearch.

Therefore, they decided to move away from Elasticsearch altogether and solely rely on Postgres.

Moving to Postgres (2019 - 2021)

In 2019, the Benchling team migrated to a new architecture for the Search System, that was solely based on Postgres.



The CRUD system remained the same as before, with Postgres being the core database.

However, searches were done as SQL queries against core tables in Postgres, so they were directly accessing the source of truth (hence no replication lag).

For [full-text search queries](#), they used [GIN Indexes](#), which stands for Generalized Inverted Indexes. An [Inverted Index](#) is the most common data structure you'll use for full text search (Elasticsearch uses an inverted index as well). The basic concept is quite similar to an index section you might find at the back of a textbook where the words in the text are mapped to their location in the textbook.

They used Postgres triggers and Celery to asynchronously update the GIN indexes for full-text search. The replication lag wasn't much of a problem here because full-text search use cases didn't typically require strong consistency (real-time syncing). They could rely instead on eventual consistency.

This setup worked great for developer productivity (no need to maintain Elasticsearch) and solved most of the replication lag issues that the team was facing.

However, Benchling experienced tremendous growth during this time period. They onboarded many new customers and users also started to use Benchling as their main data platform.

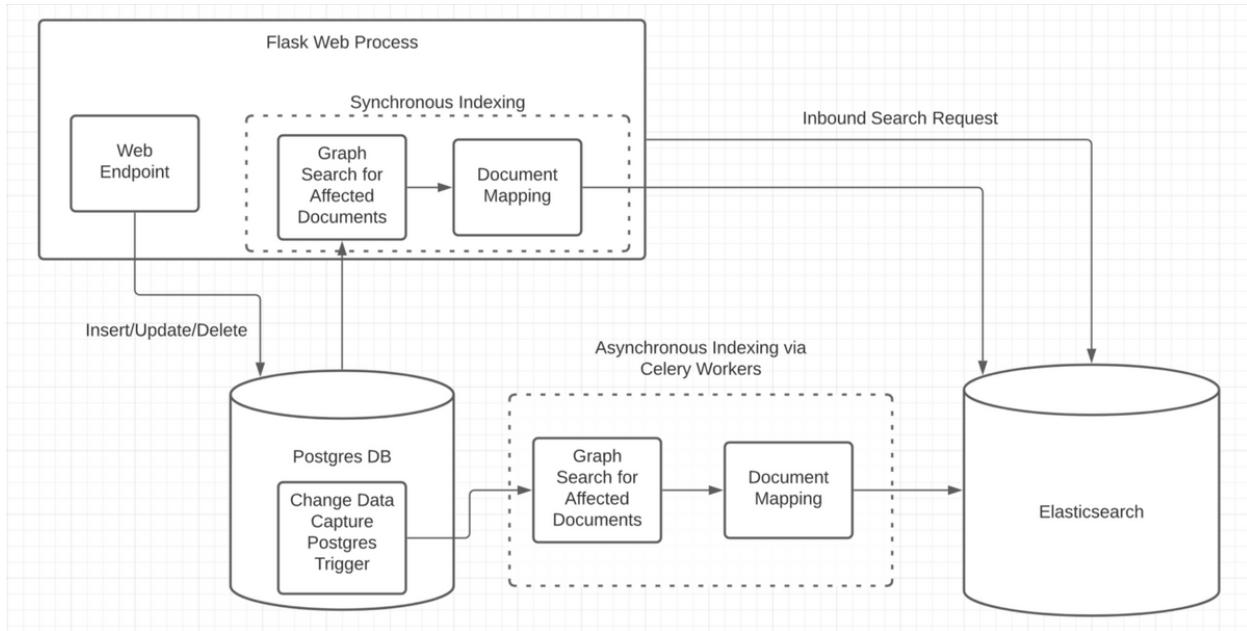
The sheer volume of data pouring into the system was orders of magnitude greater than they'd seen before.

This caused scaling issues and searches began taking tens of seconds or longer to execute. Searches were done as SQL queries against Postgres, which made heavy use of data normalization. This meant costly joins across many tables when doing a search; hence the slow reads.

Also, the Benchling team faced a lot of difficulty when trying to adapt the system to new use cases.

Building Back on Elasticsearch (2021+)

The third (and current) iteration of Benchling's search architecture is displayed below.



The team evaluated several options for search systems, but they decided to go back to Elasticsearch due to its wide industry adoption, mature plugin and text analysis system, and performant APIs.

They also made some changes to solve some of the issues from the first search system.

The data inconsistency between Postgres and Elasticsearch was the main problem, and that was due to replication lag. Benchling addressed this in the third iteration by adding the option to synchronously index the data into Elasticsearch.

With this, the CRUD actions are first copied into Elasticsearch from Postgres *and then* the user is given confirmation that the action was successful.

They addressed the write amplification issue (due to the denormalization) by tracking changes at the column level for their Postgres triggers. This greatly reduced the number of false positives that were being re-indexed.

They've also done performance testing and made some changes to their Elasticsearch cluster topology so they're comfortable that the system can handle the load of hundreds of millions of items.

For more details, you can read the full article [here](#).

How the BBC uses Serverless

The [BBC](#) is the national broadcaster for the UK and is the world's oldest and largest national broadcaster with more than 20,000 employees.

The website operates at a massive scale, with over half of the UK's population using the site every week (along with tens of millions of additional users from across the world). They have content in 44 different languages and have hundreds of different page types (news articles, food recipes, videos, etc.).

Until a few years ago, the website was written in PHP and hosted on two datacenters near London. However, the engineering team has rebuilt the website on AWS and used newer technologies like ReactJS.

The website relies heavily on [Functions as a Service \(FaaS\)](#) for scaling, specifically [AWS Lambda](#) functions.

Jonathan Ishmael is the Lead Technical Architect at the BBC, and he wrote a great series of [blog posts](#) on why the BBC chose serverless and how their backend works.

Here's a summary

Before getting into the choice of serverless, it's important to get some context about the type of workloads that the BBC website has to serve.

Traffic to the website can fluctuate greatly depending on current events, social media traffic, etc. These events can be predictable (a traffic spike during a national election) but they can also be random.

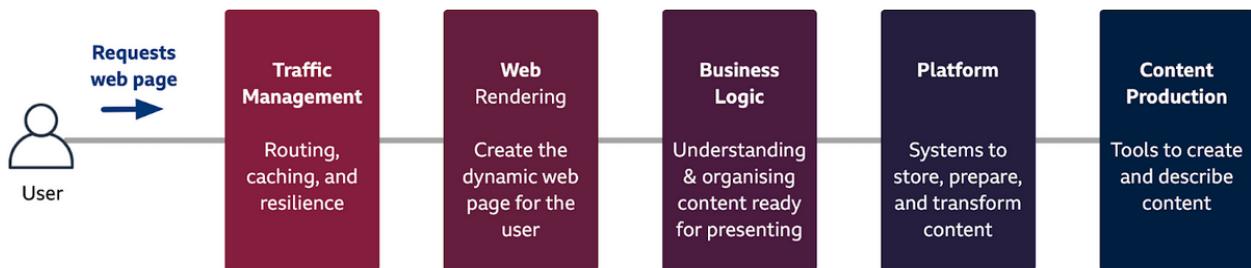
During the 2019 [London Bridge attack](#), requests for the BBC's coverage of the event resulted in a 3x increase in traffic in a single minute (4,000 req/s to 12,000 req/s). Within the next few minutes, traffic doubled again (from 12,000 req/s to 20,000 req/s).

If there's an unexpected, consequential event then the BBC's article about it can quickly start trending on social media. This brings a massive amount of traffic.

It's extremely important that the BBC website be able to quickly scale up with these traffic patterns, so that people can get access to information during an emergency.

The BBC's Backend

The BBC's backend stack can be divided into several layers.



Traffic Management Layer

All traffic to the BBC website goes to the Global Traffic Manager, which is a web server based on [Nginx](#). This layer handles thousands of requests per second and is run on AWS EC2 instances.

The layer handles caching, sanitizing requests and forwarding traffic to the relevant backend services.

The EC2 instances run with 50% reserve capacity available for handling bursts of traffic. They don't have a CPU intensive workload, so AWS [autoscaling](#) works well for high traffic events.

Web Rendering Layer

The BBC uses ReactJS for their website. They make use of React's [server-side rendering feature](#) to reduce the initial page load time when someone first visits the website. The Web Rendering layer is where the server side rendering happens.

This rendering process is quite compute-intensive, which means a ton of strain in scenarios where the amount of traffic to the BBC website shoots up (the rendering layer becomes the stress point). AWS EC2 [autoscaling](#) typically takes a few minutes to add

capacity, which is too slow for scaling the rendering layer (the amount of traffic to the site can double in less than a minute).

Therefore, the BBC relies on [AWS Lambda](#) functions for the rendering as they can scale up much faster. Approximately 2,000 lambdas run every second to create the BBC website and AWS will automatically provision more compute when there's a burst of traffic (there will be a small *cold start* time discussed below).

Business Layer

The Rendering Layer focuses solely on presentation, and it fetches data through a [REST](#) API provided by the Business Layer.

The BBC has a wide variety of content types (TV shows, movies, weather forecasts, etc.) and each one has different data / business logic.

The Business Layer is responsible for taking data from all the various BBC backend systems and transforming it into a common data model for the Web Rendering layer.

The REST API is run on EC2 instances while Lambda functions are used for the compute-intensive task of transforming data from all the different systems into a common data model.

The EC2 instances also handle intermediate caching to reduce load on the Lambda functions.

Platform and Content Production

The last two layers provide a wide range of services and tools that allow content to be created, controlled, stored and processed.



Optimizing Performance

The BBC team wanted to make sure they were optimizing their serverless functions to reduce cost and improve user experience. We'll go through a couple of the things they did.

Caching

As discussed above, the BBC has two layers that rely on serverless functions: the web rendering layer and the Business Layer.

However, they made sure to put in an intermediate caching layer between the two serverless functions to avoid the rendering functions calling any business logic functions directly.

If they didn't, then the rendering function would be sitting idle while the business logic function was working. Serverless functions are billed by [GB-seconds](#) (number of seconds your function runs for multiplied by the amount of RAM consumed), so any time spent idle is money being wasted.

The caching layers ensure that most business logic serverless functions can complete in under 50 milliseconds, reducing idle time for the rendering function.

Memory Profile

When you're working with Lambda functions, the main configurable parameter is the amount of RAM each Lambda instance has (from 128 MB to 10 GB).

The amount of memory you select will impact the available vCPUs, which impacts your response time.

Although the BBC only needed ~200 megabytes for their React app, they found that 1 gigabyte of RAM gave them the optimal price/performance point.

Cold Start Times

When you make the first request to a serverless function, your cloud provider has to copy your code bundle to a physical machine and launch a container for you. This process is referred to as a *cold start* and you'll have to wait for the Lambda function to spin up before you can get your response.

After the request is done, the cloud platform will keep the instance alive for 15-20 minutes (this differs based on provider) so any subsequent requests will not have to deal with a cold start time.

However, if you have a sudden burst in traffic, your cloud provider will have to spin up new instances to run your functions on. This means additional cold start times (although it's still faster than using EC2 autoscaling).

Factors that impact the cold start time are RAM allocation per Lambda function (discussed above), size of the code bundle, time taken to invoke the runtime associated with your code (you can write your function in Java, Go, Python, JavaScript and more), etc.

You are not charged for any of the compute that happens during the cold start process, so engineers at the BBC took advantage of this. They used that time to establish network connections to all the APIs that they needed and also loaded any JavaScript requirements into memory.

Additionally, they optimized the RAM allocated per Lambda to minimize cold start time. They found that a 512 mb memory profile increased cold start time by 3x over a 1 gigabyte memory profile, which is part of the reason why they went with 1 gb of RAM allocated.

They ended up with an average cold start time of ~250 milliseconds, with a peak of 1-2 seconds.

Performance

The BBC is running over 100 million serverless function invocations per day with 90% of the invocations taking less than 220 milliseconds (for the rendering functions).

In terms of scalability, they've been able to go from a cold system at 0 requests/sec (with everything uncached) to 5,000 requests/sec within a few minutes.

For more details, you can read the full article [here](#).

How Twitch does Chaos Engineering

Chaos Engineering is a methodology pioneered at Netflix for testing the resiliency of your system.

You simulate different failures across your system using tools like Chaos Monkey, Gremlin, AWS Fault Injection Simulator, etc. and then measure what the impact is. These tools allow you to set up simulated faults (like blocking outgoing DNS traffic, shutting down virtual machines, packet loss, etc.) and then schedule them to run randomly during a specific time window.

Chaos Engineering is meant to be done as a scientific process, where you follow [4 steps](#).

1. Define how your system should behave under normal circumstances using quantitative measurements like latency percentiles, error rates, throughput, etc.
2. Create a control group and an experiment group. In an ideal implementation, you are running experiments directly on a small portion of real user traffic. Otherwise, use the [staging](#) environment.
3. Simulate failures that reflect real world events like server crashes, severed network connections, etc.
4. Compare the difference in your quantitative measurements between the control and experimental group.

Typically, Chaos Engineering is used for measuring the resiliency of the backend (usually [service-oriented architectures](#)).

However, engineers at Twitch decided to use Chaos Engineering techniques to test their front-end. The question they wanted to answer was “*If some part of their overall system fails, how does the front-end behave and what do end users see?*”

Joaquim Verges is a senior developer at Twitch and he wrote a great [blog post](#) on Twitch’s process for chaos testing.

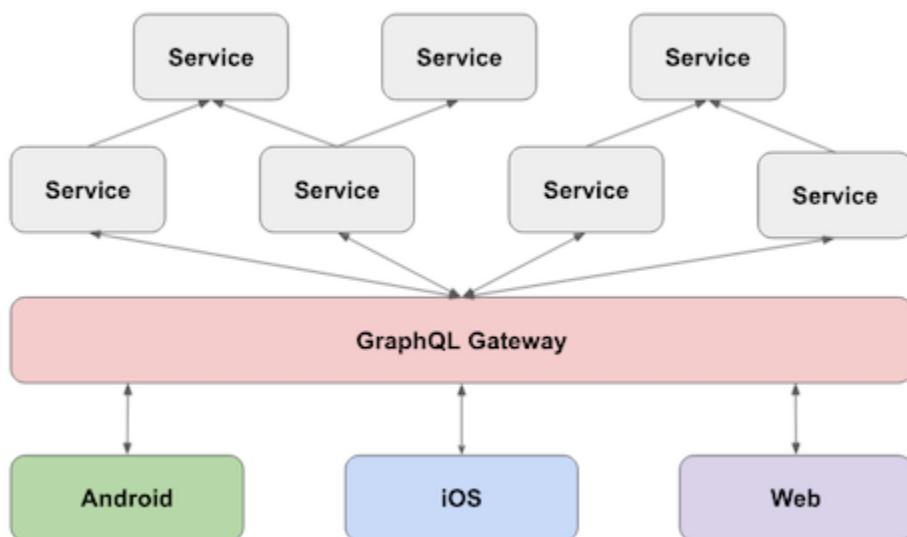
Here's a summary

Twitch is a live streaming website where content creators can stream live video to an audience. They have millions of broadcasters and tens of millions of daily active users. At any given time, there's more than a million users on the site.

Twitch uses a services-oriented architecture for their backend and they have hundreds of microservices.

The front end clients use a single [GraphQL](#) API to communicate with the backend. GraphQL allows frontend devs to use a query language to request the exact data they're looking for rather than calling a bunch of different [REST](#) endpoints.

The GraphQL server has a [resolver](#) layer that is responsible for calling the specific backend services to get all the data requested.

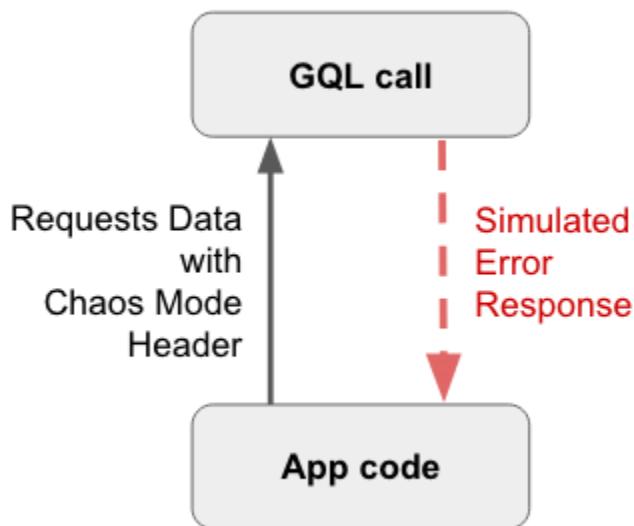


The most common fault that happens for their system is one of their microservices failing. In that scenario, GraphQL will forward partial data to the client and it's the client's job to handle the partial data gracefully and provide the best degraded experience possible.

Engineers at Twitch decided to use Chaos Engineering to test these microservice failure scenarios.

They created *Chaos Mode*, where they could pass an extra header to GraphQL calls in their staging environment. Within the header, they pass the name of the backend service that they want to simulate a failure for.

The GraphQL resolver layer will read this header and stop any call to that service.

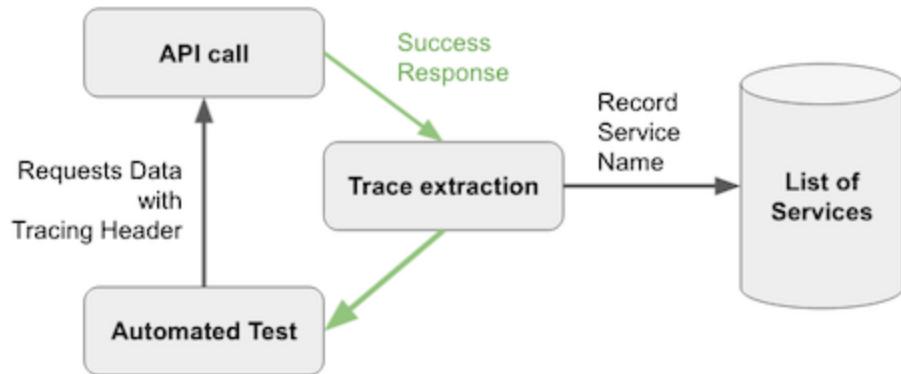


The main issue with this approach was that Twitch would need to send the name of the backend service within the GraphQL header. Therefore, they would have to maintain a list of all the various backend services to test.

Features and services are constantly changing, so manually mapping specific services to test was not scalable. They needed a way for the test suite to “discover” the services that they should be simulating failures for.

To solve this, Twitch added a debug header in their GraphQL calls which enabled tracing at the GraphQL [resolver](#) layer. The resolvers record any method call done to internal service dependencies, and then send the information back to the client in the same GraphQL call.

From there, the client can extract the service names that were involved and use that as input for the Chaos Testing suite.



Visualization

Twitch has many end-to-end tests for all their clients that test the various user flows (navigating to a screen, logging in, sending a chat message, etc.)

They try each of these tests with all of the Chaos Mode microservice failures and see whether the test was successful. Then, they aggregate all the Chaos Mode test results for each user flow and use that to calculate a resilience score for that particular user action test.

Resilience scores are displayed on a Dashboard where it's easy to see any anomalies in performance. They run Chaos Mode tests every night for their Android, iOS and Web clients.

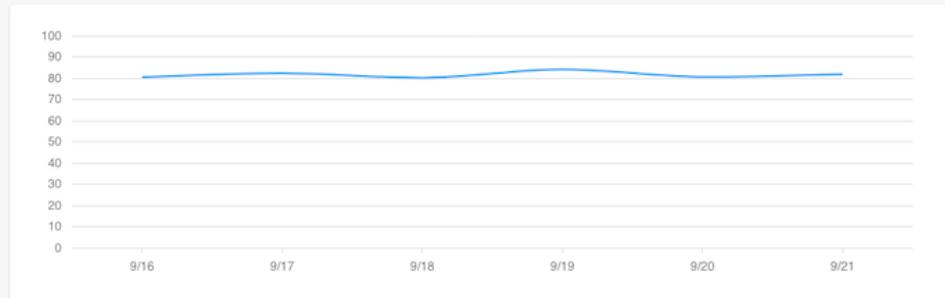


ANDROID

IOS

WEB

Android Resilience Trends



	9/16	9/17	9/18	9/19	9/20	9/21
Client Resilience	80.440%	82.336%	80.175%	84.184%	80.631%	81.877%
Ability to login	90.909%	86.364%	86.364%	90.909%	90.909%	90.909%
Ability to make a living	75.862%	84.375%	75.862%	87.500%	75.000%	79.310%
Ability to interact with a community	63.636%	75.000%	73.913%	72.222%	73.913%	79.167%
Ability to discover content	88.462%	82.609%	77.778%	86.957%	80.000%	75.000%
Ability to watch a broadcast	83.333%	83.333%	86.957%	83.333%	83.333%	85.000%

Next Steps

Twitch has been able to use this testing tool to boost resilience across all their clients.

Next they want to add the ability to test secondary microservices (services that are called from another service rather than just testing services that are called directly from the GraphQL resolver layer).

They also want to add the ability to simulate failures for multiple services at once.

For more details, you can read the full blog post [here](#).

How PayPal uses Graph Databases for Fraud Prevention

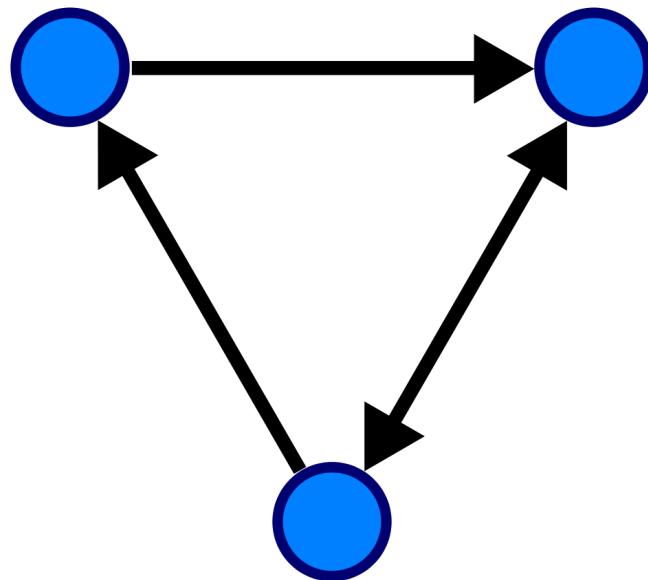
PayPal is a fintech company that lets users transfer money online. They have over 400 million active consumers and merchants on the platform and they process thousands of payment transactions every minute.

Detection and prevention of fraud is one of the biggest problems fintech companies have to deal with and PayPal is no exception.

In order to do this, PayPal relies on a graph database. Quinn Zuo is the head of AI/ML Product Management at PayPal and he wrote a great [blog post](#) on how PayPal does this.

Here's a summary

A [graph](#) is a collection of nodes (vertices) and relationships (edges) between those nodes.



A graph database management system (graph database) gives you a durable way to store your graph along with an interface to easily perform create/read/update/delete (CRUD) operations on the graph.

Relationships are first class citizens in graph databases, so traversing through nodes and computing various graph algorithms ([PageRank](#), [Connected Components](#), etc.) is much easier to implement compared to doing it with SQL queries on a relational database. It can also be significantly faster than a relational database depending on how the graph database is built.

You can view a comparison of querying for data between SQL and Cypher (Neo4J's graph query language) [here](#).

If you have a graph database with data about actors and the movies they were involved with, then an SQL query for all the directors of Keanu Reeves movies might look like this...

Sql

```
SELECT director.name, count(*)
FROM person keanu
JOIN acted_in ON keanu.id = acted_in.person_id
JOIN directed ON acted_in.movie_id = directed.movie_id
JOIN person AS director ON directed.person_id = director.id
WHERE keanu.name = 'Keanu Reeves'
GROUP BY director.name
ORDER BY count(*) DESC
```

A Cypher query would be much shorter and easier to interpret...

Cypher

```
MATCH (keanu:Person {name: 'Keanu Reeves'})-[:ACTED_IN]->(movie:Movie),
      (director:Person)-[:DIRECTED]->(movie)
RETURN director.name, count(*)
ORDER BY count(*) DESC
```

Properties

When looking at graph database technologies, there are two properties you should be examining: *the underlying storage* (native vs. non-native storage) and *the processing engine* (native vs. non-native processing).

Underlying Storage

This is the underlying structure of the database that contains the graph data.

It can be either *native* or *non-native*. Native graph storage means that it's been built specifically for storing graph-like data, which means more efficiency when running graph queries. You can see this with graph databases like [Neo4j](#). For non-native storage, the graph database will serialize the graph data into relational, key-value, document-oriented or some other general-purpose data store.

The benefit of non-native graph storage is that you can build your graph database on a battle-tested backend like Postgres or Cassandra where the scaling characteristics are well understood. You can take advantage of a Graph API without having to rebuild all the sharding, replication, redundancy, etc.

PayPal uses [Aerospike](#) as the underlying storage for their graph database. Aerospike is an open source, distributed, key value database.

The Processing Engine

The processing engine runs database operations on your graph, and can be split into native or non-native processing.

The key difference between native and non-native processing is *index-free adjacency*. Index-free adjacency means that your graph doesn't have to work with a [database index](#) to hop from any node to its neighboring nodes. Each node has direct addresses to all of its neighboring nodes. Native graph processing means using index-free adjacency.

On the other hand, if you're using a relational database as the underlying storage and you need to first check a [database index](#) to find the location of a neighboring node, then you do not have native graph processing.

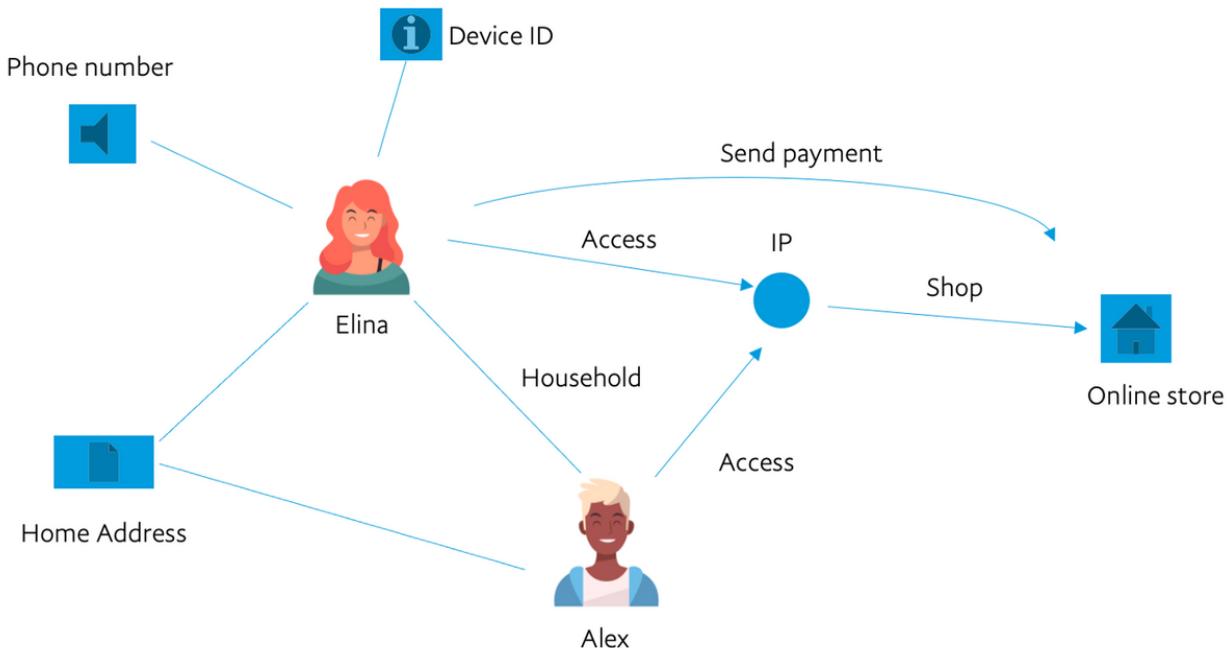
[Here's](#) a great article that delves into index-free adjacency.

Graph Database

PayPal has a two-sided network with buyers and merchants who are sending each other transactions.

They encode this network as a graph with buyers/sellers modeled as vertices in the graph.

Edges are connections between the vertices. Examples of potential connections are sending a payment, sharing the same IP, having the same home address, etc.



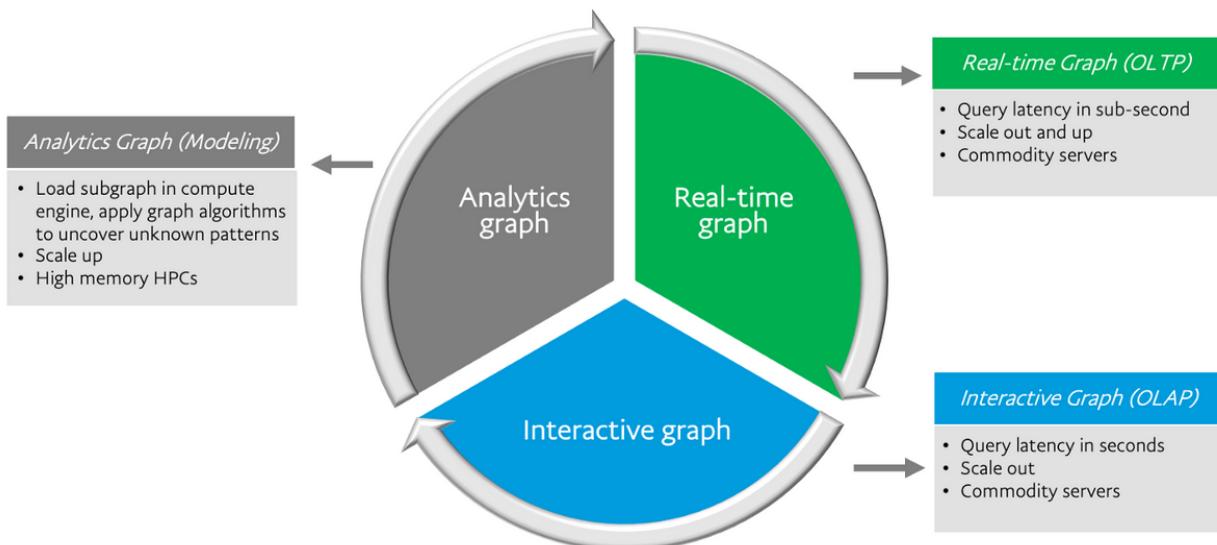
An Overview of PayPal's Graph Platform

PayPal's graph platform is an integrated platform that consists of real-time, interactive, and analytics graph technology stacks.

The real-time graph platform will return graph query results very quickly (sub-second). The returned query results can be used in machine learning models for immediate fraud prevention. If a malicious actor tries to create a new PayPal account after getting banned, the real-time graph platform can help identify that user and block his account right after he creates it.

The Interactive graph platform serves use cases where the query latency can be within a few seconds or minutes. This is useful for graph visualization and is suitable for investigations done by PayPal's fraud-prevention teams.

The analytics graph platform is used to uncover unknown patterns using graph algorithms and training graph ML models. It's built on [HPCs](#) so that training and algorithms can be run quickly whereas the interactive and real-time platforms are both built on commodity servers.



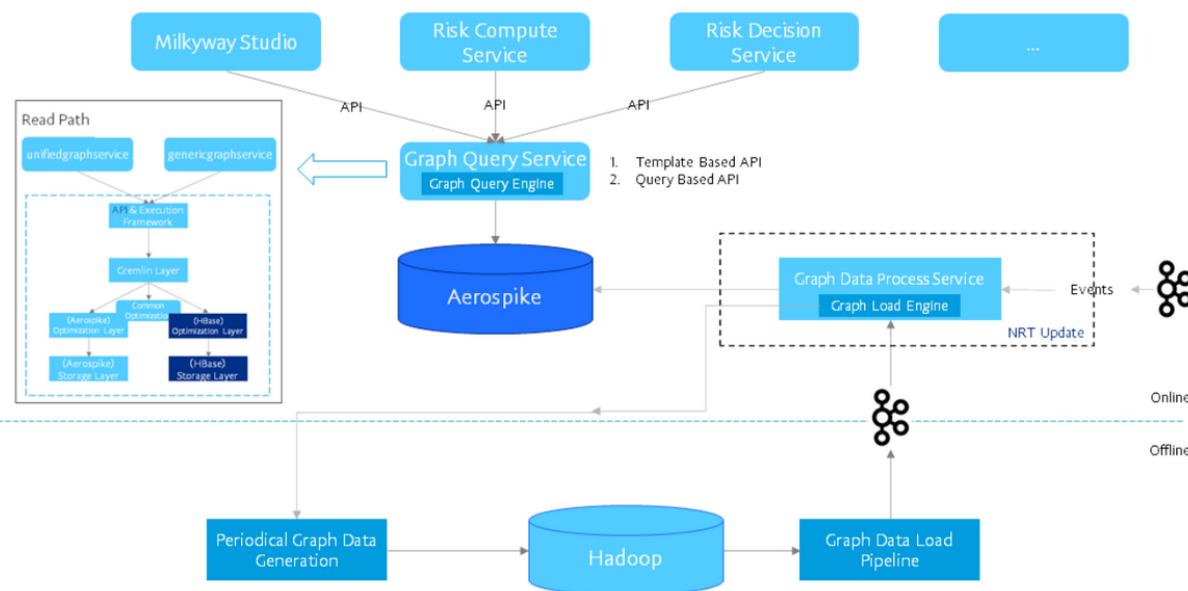
Real Time Graph Database

The real time graph platform is used to query the graph database and identify potential fraudulent activities immediately.

Some of the core requirements for PayPal's Real Time Graph Platform are

- Customizable high performance graph query APIs
- Sub-second level query latency and optimization
- Seconds level data freshness (near real-time update)
- Horizontal scalability with fault tolerance
- Million queries per second throughput
- Flexible data backfill from offline to online

Here's the architecture for the Real-Time Graph Stack. You can view a larger image [here](#).



The storage backend for the database uses Aerospike and there's a Write Path and a Read Path built around it to perform CRUD operations on the database.

Write Path

For the write path, offline data loading (in the lower part of the image above) and event-based data updates are abstracted into a single Graph Data Process Service.

The offline channel is set up for loading snapshots of the data and supports daily or weekly updates.

Event-based, near real-time data comes from a variety of production data services at PayPal. These data sources have been abstracted as events/messages in Kafka. The Graph Data Process Service consumes those messages to create new vertices and edges in the graph database.

Read Path

The Graph Query Service is responsible for handling reads from the underlying Aerospike data store. It provides template APIs that the upstream services (for running the ML models) can use.

Those APIs wrap [Gremlin](#) queries that run on a Gremlin Layer. Gremlin is an open source graph query language that can be used for OLTP and OLAP traversals. It's part of Apache [TinkerPop](#), which is a popular graph computing framework.

The Gremlin layer converts the queries into optimized Aerospike queries, where they can be run against the underlying storage.

For more details on PayPal's Graph viewer and Graph embeddings, you can read the full article [here](#).

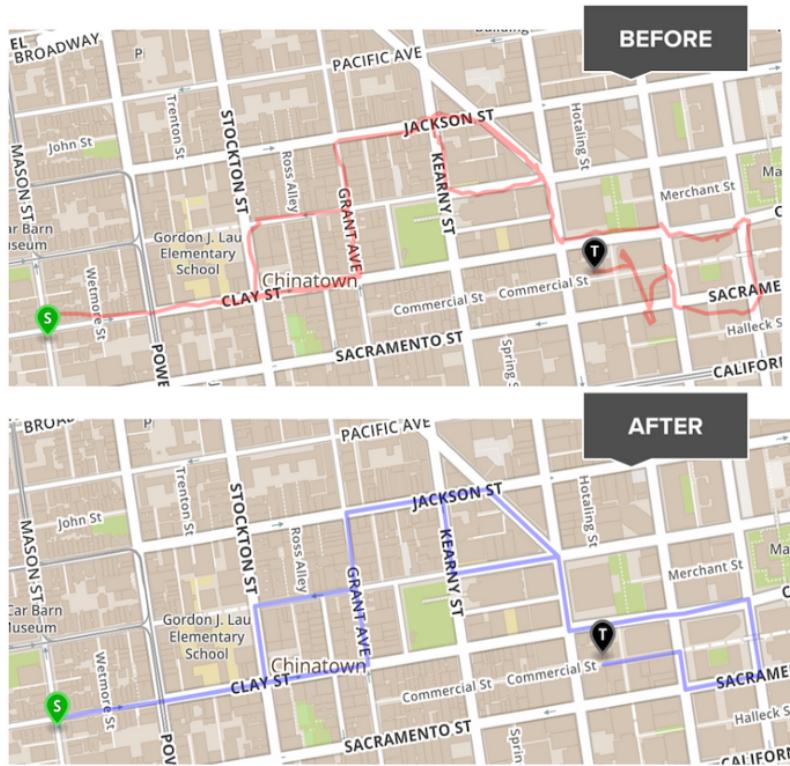
Client Side Localization at Lyft

Lyft is a ride-sharing app that allows riders to connect with drivers (similar to DiDi, Uber, Grab, Ola Cabs, etc.). The company is the second-largest ridesharing app in the US with 29% market share and close to 20 million users.

In order to help the rider and driver connect, the Lyft app shows both users a map with the real time location of the other person.

Lyft relies on GPS data from both the riders and drivers mobile devices for the real-time position, however GPS signals are notoriously noisy and unreliable. Relying on GPS signals alone would mean inaccurate real-time positioning of the rider and driver, resulting in a poor user experience.

The Mapping team at Lyft solves this issue by using map data to more accurately localize the driver/rider. This reduces the space of locations to just the roads and makes it much easier to run map matching algorithms (match the user to the correct position on the map). You can read about the map matching algorithms that Lyft uses [here](#).



Previously, these localization systems were run server-side. Map data was stored on Lyft servers and their map matching algorithms would be run server side. However, in 2021 Lyft made the transition to client side localization.

Karina Goot is an Engineering Manager at Lyft and she wrote a great [blog post](#) on this transition.

Here's a summary

Benefits of Client-side Localization

There are quite a few benefits from shifting localization to the client rather than running it server-side.

- Network Benefits - Cell network availability is not guaranteed. Putting map data on the client and having localization algorithms running on the user's phone means better localization when the user is can't connect to Lyft servers (in a tunnel or in areas with poor connection)
- Efficiency and Performance Benefits - Moving the high-cost computation from the server to the client will simplify the server-side architecture, reduce hosting costs and also lower server-to-client latency.
- Driver Safety Features - Running localization client side means that map data has to be on the driver's phone. Lyft can later use that map data to add in additional features to the UI like symbols for traffic lights, stop signs, speed limits, etc.

The drawback to putting localization client-side is that it is extremely constrained in memory and latency requirements.

Lyft cannot put too much map data on the client or that will cause the Lyft app to take up too much user storage. They also can't send all the map data through the network as downloading data while on cellular is expensive and slow.

The Lyft Engineering team had to navigate these technical limitations and designed the client localization system around them.

Designing the System

They broke the project down into three main components.

1. Generating Lightweight Map Data
2. Client Platform Networking Layer
3. C++ Core Library for localization

Generating Lightweight Map Data

Lyft has to send map data to the client devices without taking up too much user storage, so they can only include data about the user's local area. This means changing up the map data format, how it's generated and how it's served.

To do this, they used the [S2 Geometry](#) library. The S2 library was developed at Google and made [open source in 2017](#). It represents all data on a three-dimensional sphere, which models map data better than traditional geographic information systems that represent data in two dimensions.

The team divided the entire LyftMap into small chunks of [S2 Cells](#). When the client tries to download map data from the server, it specifies the cell id of the S2 cell and the map version. The server will then return the map data serialized as S2 Cell Elements.

The client will download the necessary cells based on the user location and dynamically build the road network graph in memory.

Client Platform Networking Layer

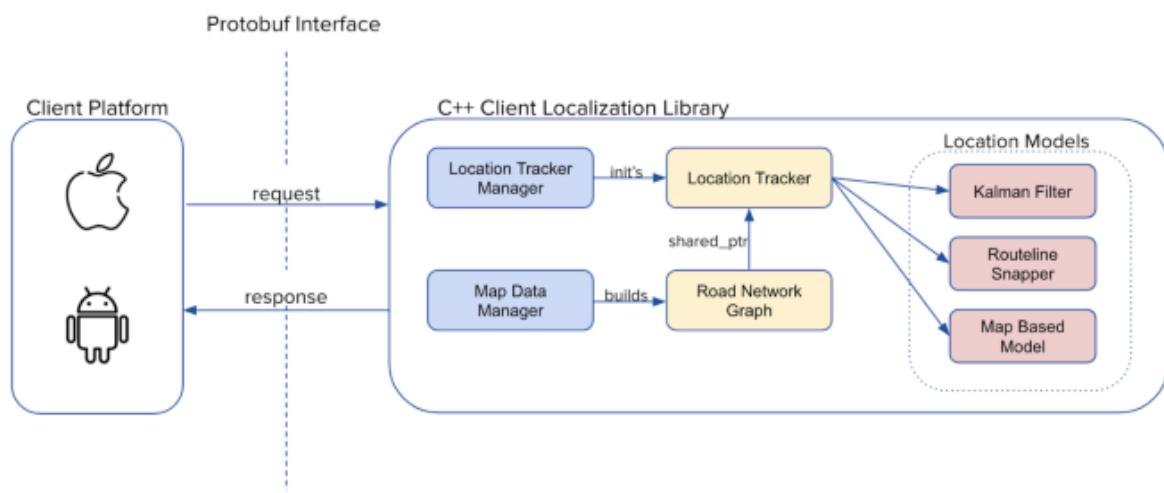
Lyft created a backend service called MapAttributes, that reads the map elements data from DynamoDB based on a [geospatial index](#). The S2 library uses the [Quadtree](#) data structure for geospatial indexing. [Here's](#) a great blog post on how S2 does indexing if you'd like to learn more. These map elements are serialized and converted to S2 Cell Elements.

To reduce latency and increase reliability, Lyft also uses AWS CloudFront's CDN to cache the MapAttributes responses. CloudFront will return the results from the CDN on a cache hit without calling the MapAttributes service.

Core Library Functionality

Once the client fetches the desired map cells from the server, it passes this information through to a C++ localization library on the client.

This library uses Lyft's [Map-Based Models](#) for localization.



Lyft drivers generally operate in a single service area so their locality is highly concentrated. This causes duplicate downloads of the same data, leading to unnecessarily high network data usage.

To solve this, engineers added an in-memory [SQLite](#) caching layer directly in C++. They used SQLite because of its simplicity and native support on client platforms.

With this cache, they can store the highest locality map data for each driver directly on-device. By persisting the map data on disk, they can store data across sessions and only have to refresh the cache when the underlying map data changes.

Based on data analysis of driving patterns, the Lyft team found that they can achieve a high cache hit rate for the vast majority of Lyft drivers with only 15 megabytes of on-device data.

Avg driver count of cells accessed/day	Avg driver count of cells accessed/month	Percent drivers using > 1000 cells/month	Average cell size in kilobytes
71	389	6%	15 KB

Results

In order to track the success of the project, Lyft looked at how often mobile clients have map data and what the latency of the map matching system was.

They found > 99% on-device map data availability among drivers and sub 10 ms latency for 99% of map matching computations.

With this project, drivers and riders now have significantly better map localization in the Lyft app.

You can read more [here](#).

How Airbnb rebuilt their Payments System

In 2020, Airbnb migrated from a Ruby on Rails monolith to a service-oriented architecture. This shift helped Airbnb increase developer velocity as the engineering team grew to thousands of globally distributed developers and millions of lines of code.

However, this change also brought multiple challenges that the team had to deal with. Data was now scattered across many different services so aggregating information in the presentation layer was complicated, especially for complex domains like payments. Getting all the information on fees, currency fluctuations, taxes, discounts and more meant that there were far too many different services to call.

Airbnb addressed this by adding a [service mesh](#) to provide a unified endpoint to the client services in the presentation layer. A service mesh is a layer of proxy servers added to facilitate communication between microservices. You can also add observability, security and reliability features into the service mesh rather than at the application layer (in the microservices).

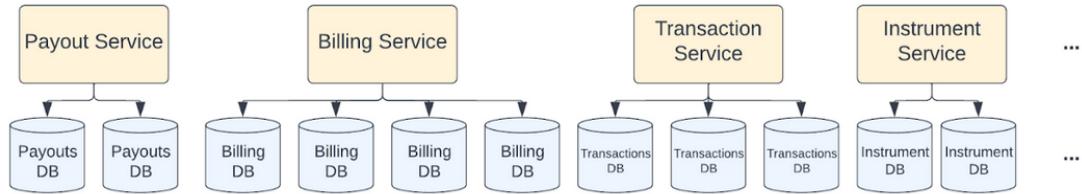
Ali Can Göksel is a senior software engineer at Airbnb and he wrote a great [blog post](#) on how Airbnb re-architected their Payments layer to incorporate Viaduct, a service mesh built on GraphQL.

Here's a summary

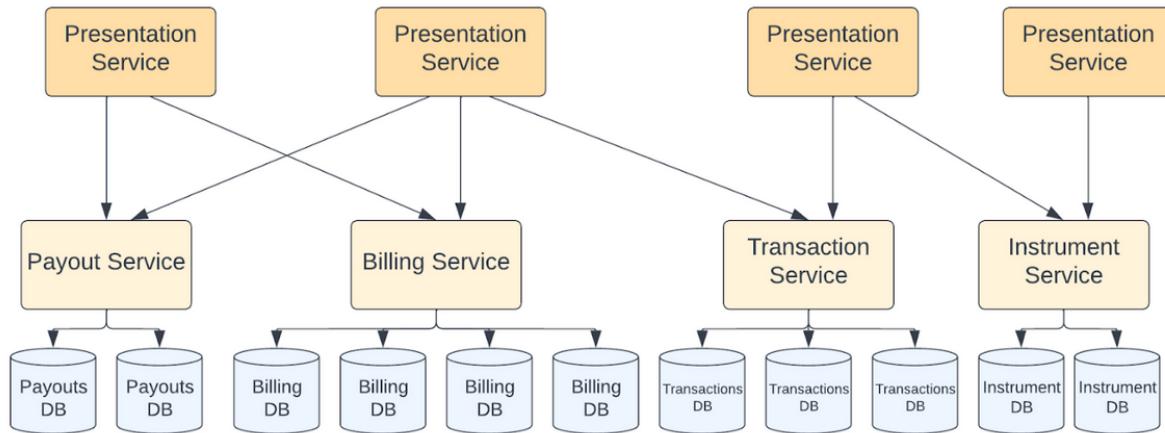
During their migration to a services-oriented architecture, Airbnb broke up their payments layer into multiple services.

This helped provide

- A clear boundary between different payment services. This enabled better domain ownership and faster development iteration.
- Better data separation into the different domains. Data was kept in a [normalized](#) shape (where you reduce data redundancy). This resulted in better correctness and consistency.



The downside of this is that the clients in the presentation layer now had to integrate with multiple payment services to fetch all the required data. They had to look into multiple services and read from even more tables than prior to the services migration.



This resulted in 3 main challenges

1. The system was hard to understand - Client teams needed a deep understanding of the Payments domain in order to find the right payments services to gather all the data they needed. This reduced developer velocity for those client teams and also meant engineers on the payment side needed to provide continuous guidance and consultation.
2. The system was difficult to change - When the payments team had to update their APIs, they had to make sure that all dependent presentation services adopted these changes.

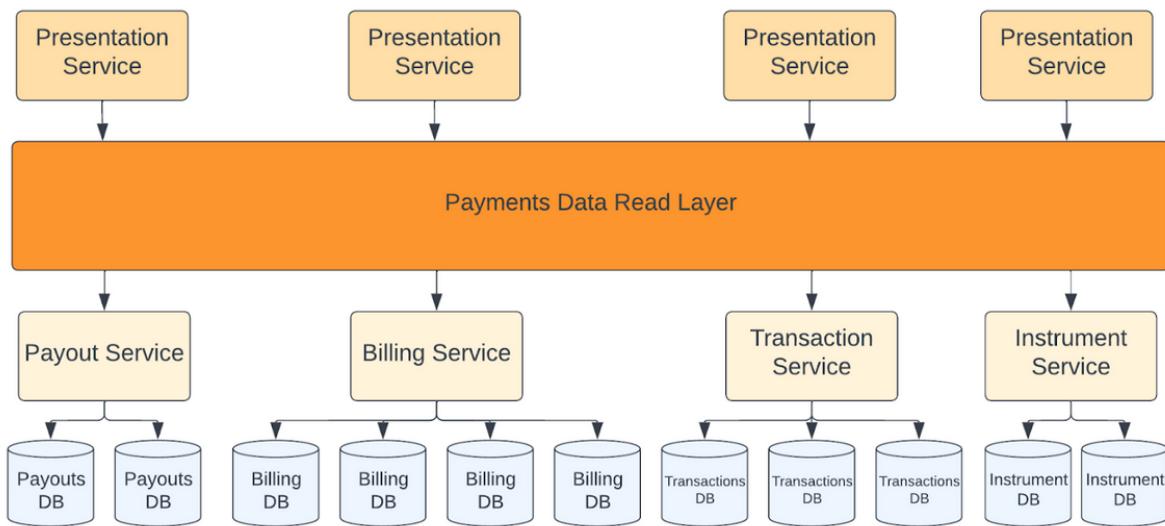
- Poor performance and scalability - The technical quality of the complex read flows was not up to standards. Aggregating payment-related data for large hosts with thousands of yearly bookings was creating a scaling problem.

Unified Entry Point

Airbnb addressed these challenges by adding a Payments Data Read Layer that acted as a service mesh. To do this, they used [Viaduct](#), which is built on GraphQL.

With this, clients can query the layer for the data entity instead of having to identify dozens of services and their APIs.

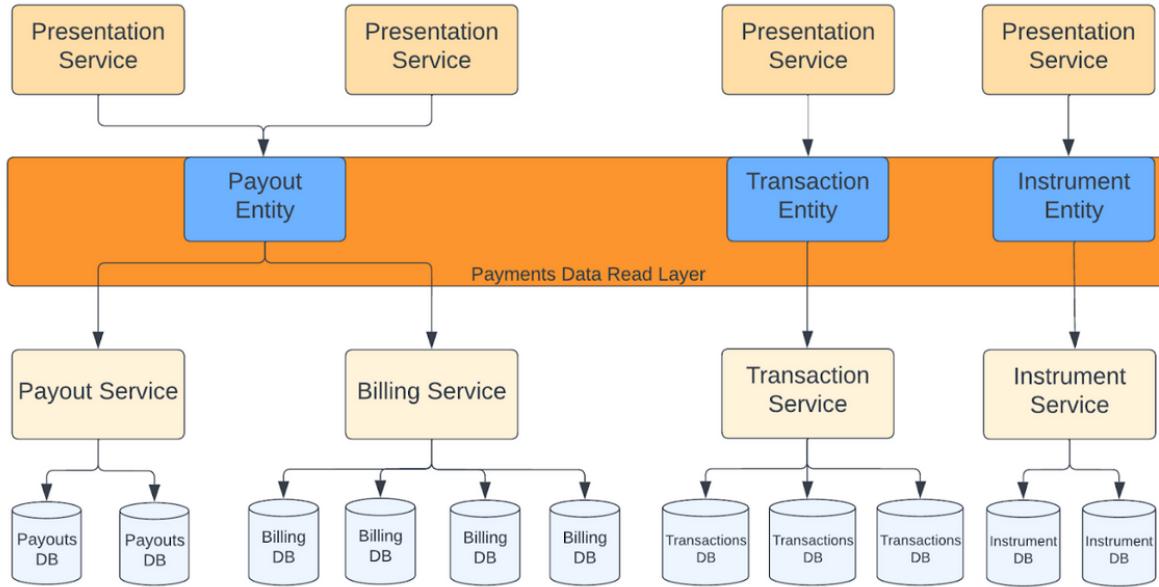
This greatly reduced the number of APIs that needed to be exposed.



However, just using a single entry point doesn't resolve all the complexity. Their payments system has 100+ data models, and exposing all of them from a single entry point would still be overly complex for client engineers.

To simplify this, they created higher-level domain entities to further hide internal payment details. They made fewer than 10 high level entities, so it became much easier for client teams to find the data they wanted. Also, Airbnb could now make changes to

internal Payment business logic while keeping the entity schema the same so they wouldn't have to rewrite any code for the consumer of the payment data read layer.



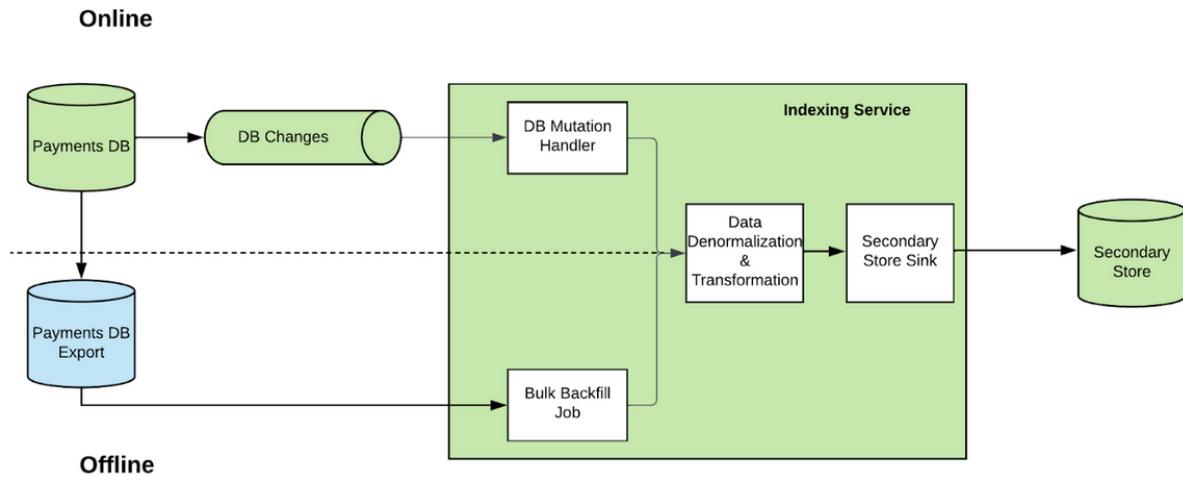
Improving Performance

As stated earlier, one of the challenges in the previous system was poor performance and scalability. The complex read flows of fetching the data from all the different services caused too much latency, especially for large hosts.

The core problem was reading and joining many different tables and services while executing client queries. To solve this, Airbnb added secondary denormalized Elasticsearch indices to serve as read replicas.

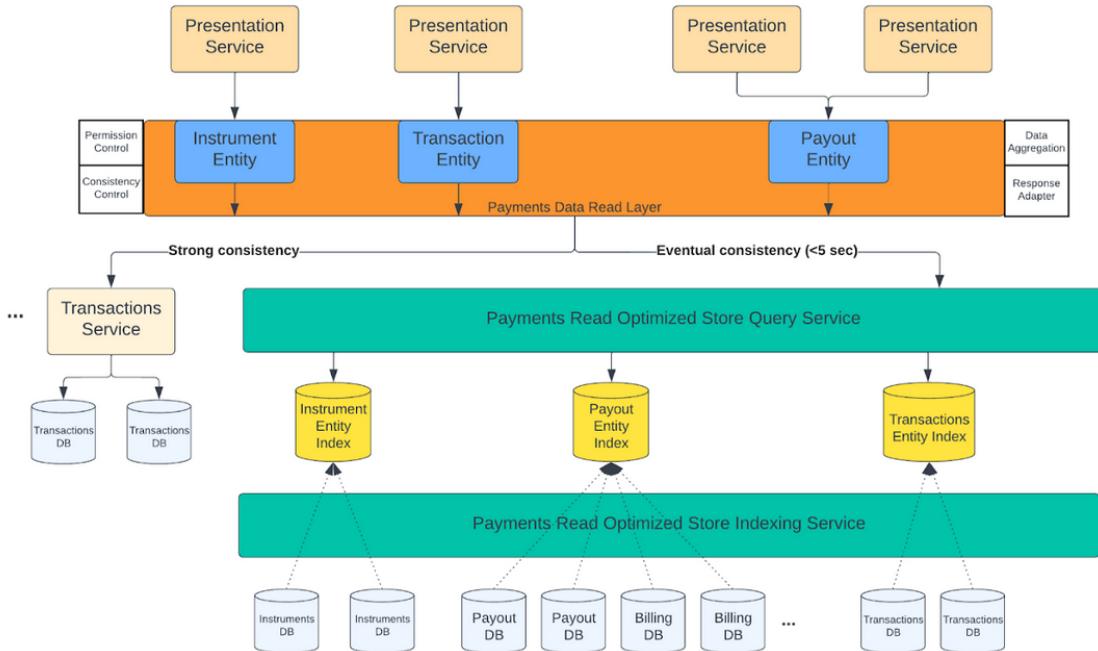
This moves the expensive operations from query time to ingestion time. Instead of doing lots of joins during a query, the data has to be written to the replicas during ingestion. It also sacrifices data consistency due to replication lag.

They created a system where real-time data could be written to the secondary store via database change data capture mechanisms and historical data could be written through daily database dumps. They were able to reliably achieve less than 10 seconds of replication lag.



Airbnb denormalized the data from the tables into Elasticsearch. This greatly reduced the touchpoints of the query and also made pagination and aggregation much more efficient.

After combining all of the above improvements, their new payments read flow looked like the following



The presentation service would query one of the domain entities for the data it was looking for. If the request didn't need strong consistency, then it could go to the Elasticsearch index for that entity. Otherwise, it would go to the entity's master database. The indexing service would make sure the Elasticsearch replicas are updated with new changes from the master.

This shift to denormalization resulted in up to 150x latency improvements and improved reliability to 99.9%.

For more details, you can read the full post [here](#).

Dropbox's Asynchronous Task Framework

Dropbox is a file hosting and sharing company with over 700 million registered users. Hundreds of thousands of companies also rely on Dropbox for their business needs like storing and sharing documents, team collaboration, etc.

Like many other companies, Dropbox relies on an asynchronous task manager (called Asynchronous Task Framework or ATF) to manage and run async tasks. When you remove a file on your Dropbox account, the UI may show that the file was moved; but behind the scenes, an async task was created to delete the file on all the database replicas.

ATF (Asynchronous Task Framework) serves more than 9000 async tasks scheduled per second, and more than 30 teams at Dropbox make use of the framework.

Arun Sai Krishnan is a Software Engineer at Dropbox, and he wrote a great [blog post](#) on design goals of ATF and the architecture behind it.

Here's a summary

ATF allows Dropbox engineers to schedule async tasks on-demand through a callback-based architecture. Developers can define [callback functions](#) and then schedule ATF tasks that execute these callbacks.

The callback functions are called lambdas, and developers can write lambdas to execute async tasks like sending out an email to a user.

When an engineer wants to execute a lambda, they can submit it to the ATF. This creates a task, which is just a unit of execution of a lambda (similar to how a process is a unit of execution of a program).

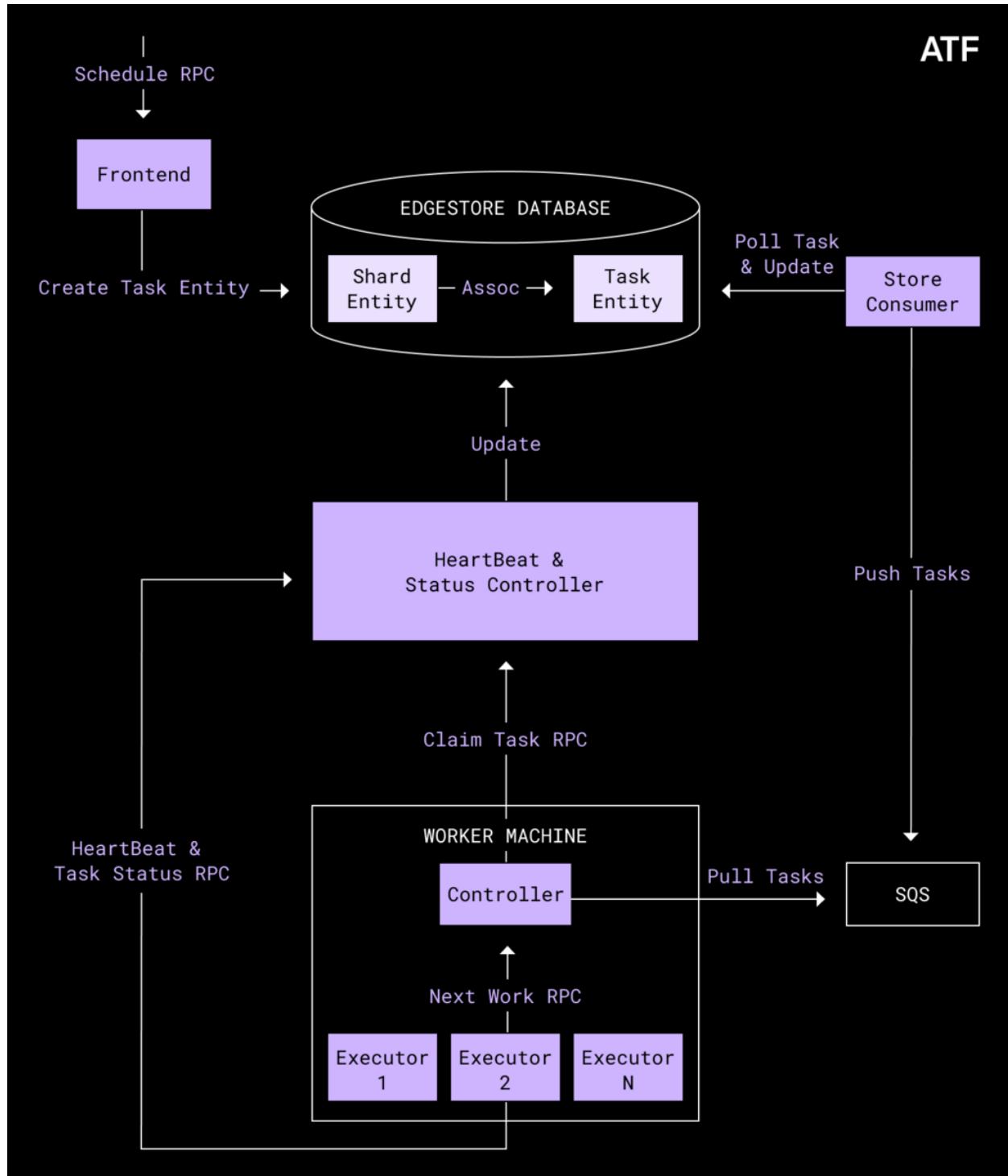
ATF supports features like

- Task Scheduling - schedule the task to execute at a specific time.

- Priority Based Execution - tasks with higher priority get executed before tasks with lower priority.
- Task Status Querying - clients can query the status of a scheduled task.

ATF Architecture

Here's the Architecture for ATF



ATF consists of the following components

- Frontend - Clients can schedule tasks using [remote procedure calls](#) (RPC). Dropbox uses gRPC with an in-house built RPC framework called [Courier](#).
- Task Store - The frontend accepts tasks and stores them in the task store. This can be any generic data store that has indexed querying capability. Dropbox uses their in-house metadata store called [Edgestore](#). It's built on top of MySQL.
- Store Consumer - The store consumer is a service that will periodically poll the task store to find tasks that are ready for execution. It pushes these tasks onto the right queue.
- Queue - Dropbox uses AWS [Simple Queue Service \(SQS\)](#) to queue the tasks. Worker machines will pull tasks off the SQS queues.
- Controller - Worker machines consist of a controller and multiple executors. The controller process is responsible for polling tasks from SQS queues and pushing them onto process local buffered queues. Then, it serves these tasks from its local queue as a response to Next Work RPC requests.
- Executor - The executor is a process with multiple threads that is responsible for the actual task execution. It gets tasks from the Controller by polling for work from the Controller by sending Next Work RPC requests.
- Heartbeat and Status Controller (HSC) - The HSC serves RPCs for status updates during task execution and setting task status in the task store after execution.

ATF provides the following system guarantees

- At-least Once Task Execution - tasks will be executed *at least once*. The ATF will try and retry tasks until they complete execution or reach a fatal failure state. This means that a task may get executed multiple times, so developers have to ensure that their lambda logic is **idempotent** (can be run multiple times without changing the result).
- No Concurrent Task Execution - The ATF system guarantees that at most one instance of a task will be actively executing at any given time, so developers can write their callback logic without designing for concurrent execution of the same task from different workers. Before a task starts execution, it will be marked with a state of “Claimed” so it doesn’t get assigned to another worker machine.
- Delivery Latency - 95% of tasks begin execution within 5 seconds from their scheduled execution time. The store consumer polls for ready tasks once every two seconds. This polling frequency can be configured to change the task delivery latency.
- 3 Nines Availability - The ATF service is 99.9% available to accept task scheduling requests from any client.

For more details on ATF’s ownership model, task lifecycle and data model, you can read the full article [here](#).

The Architecture of Facebook's distributed Key Value store

ZippyDB is a strongly consistent, distributed key-value store built at Facebook. It was first deployed in 2013 and it serves many use cases like storing product data, keeping track of events and storing metadata for a distributed file system.

Because it serves a variety of different use cases, ZippyDB offers users a lot of flexibility with the option to tune durability, consistency, availability and latency to fit the application's needs.

Sarang Masti is a software engineer at Facebook and he wrote a great [blog post](#) about the design choices and trade-offs made in building the ZippyDB service.

Here's a summary

Before ZippyDB, various teams at Facebook used [RocksDB](#) to manage their data. RocksDB is a fork of Google's [LevelDB](#) with the [goal of improving performance for server workloads](#).

However, the teams using RocksDB were each facing similar challenges around consistency, fault tolerance, failure recovery, replication, etc. They were building their own custom solutions, which meant an unnecessary duplication of effort.

ZippyDB was created to address the issues for all these teams. It provides a highly durable and consistent key-value data store with RocksDB as the underlying storage engine.

Data Model

ZippyDB supports a simple key-value data model with APIs to get, put and delete keys. It supports iterating over key prefixes and deleting a range of keys, similar to what you get with RocksDB.

They also have [TTL](#) (Time to live) support for ephemeral data where clients can specify the expiry time for a key-value pair.

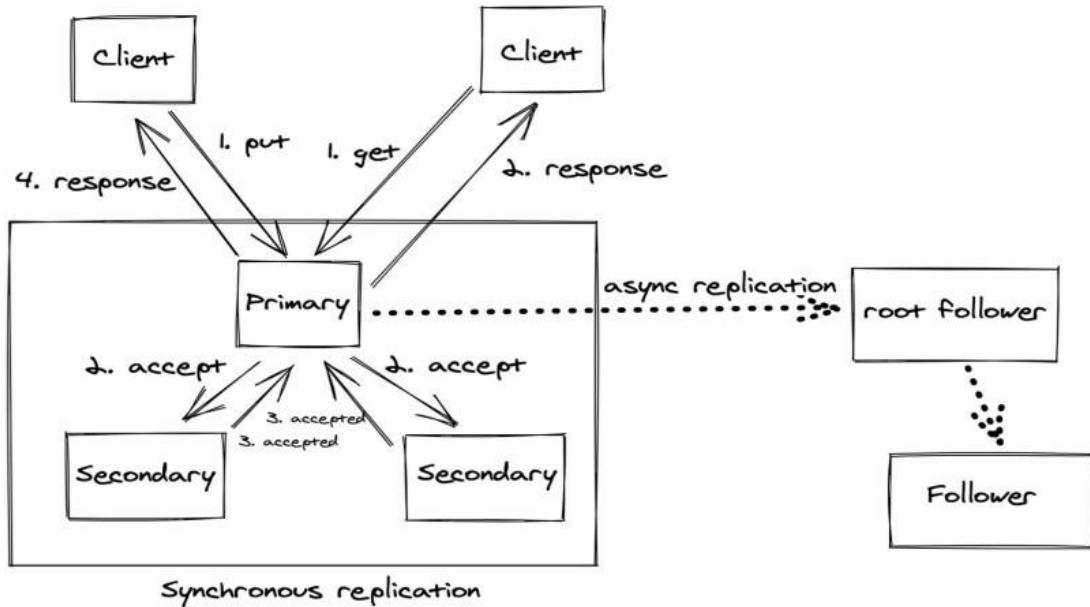
Architecture

The basic unit of data management for ZippyDB is a shard, where each shard consists of multiple replicas that are spread across geographic regions for fault tolerance. The replication is done with either [Paxos](#) or async replication (depending on the configuration).

Within a shard, a subset of the replicas are configured to be part of the Paxos quorum group, where data is synchronously replicated between those nodes. The write involves persisting the data on a majority of the Paxos replicas log's (so Paxos' consensus algorithm will return the new write) and also writing the data to RocksDB on the primary. Once that's done, the write gets confirmed to the client, providing highly [durable](#) writes.

The remaining replicas in the shard are configured as followers. These receive data through asynchronous replication. These replicas handle low-latency reads with the tradeoff being that they have worse consistency.

The quorum size vs. the number of follower replicas is configurable, and it lets a user strike their preferred balance between durability, write performance, read performance and consistency. We'll talk about ZippyDB's consistency in the next section.



The optimal assignment of shards to servers depends on the shard load, user constraints, etc. It's handled by another Facebook service called [ShardManager](#). ShardManager handles monitoring shards for load balancing, failure recovery, etc.

Each shard has a size of 50 - 100 gigabytes and is split into several thousand microshards which are then stored on different physical servers. This additional layer of abstraction allows ZippyDB to reshuffle the data without any changes for the client.

ZippyDB maps from microshards to shards with two types of mapping: Compact mapping and Akkio mapping.

Compact mapping is used when the assignment is fairly static and mapping is only changed when there is a need to split shards that have become too large or hot.

Akkio mapping is more involved and tries to optimize microshard placement to minimize latency. You can read about how Akkio mapping works [here](#).

Consistency

ZippyDB provides configurable consistency and durability levels as options in the read/write APIs. This allows users to make durability, consistency and performance trade-offs dynamically on a per-request level.

By default, a write involves persisting the data on a majority of the Paxos replicas' logs and also writing the data to RocksDB on the primary before confirming the write to the client. Persisting the write on a majority of the Paxos replicas means that the Paxos Quorum will return the new value.

However, some applications need lower latency writes so ZippyDB also supports a fast-acknowledge mode where writes are confirmed as soon as they are enqueued on the primary for replication. This means lower durability.

For reads, the three most popular consistency levels for ZippyDB are

- Eventual
- Read-your-writes
- Strong

Eventual - This is a much stronger consistency level than what's typically described as eventual consistency. ZippyDB ensures that reads that are served by follower replicas aren't lagging behind the primary/quorum beyond a certain configurable threshold. Therefore, it's similar to something like Bounded Staleness that you might see in Azure's [CosmosDB](#).

Read-Your-Writes - The client will always get a replica that is current enough to have any previous writes made by this client. In order to implement this, ZippyDB assigns a monotonically increasing sequence number to each write and it'll return this number in response to a client's write request. The client can use their latest sequence number

when sending read requests to ensure that it gets a replica that's up to date on all the client's past writes.

Strong - The client will see the effects of the most recent writes. This is done by routing the read requests to the primary.

For more details on how ZippyDB implements transactions and conditional writes, you can read the full article [here](#).

Challenges with Distributed Systems

When you’re working at a massive scale, you’ll usually have to resort to [horizontal scaling](#) to scale the system up. This means working with a distributed system and dealing with all the ensuing challenges.

Jacob Gabrielson is a VP & Distinguished Engineer at Oracle and was previously a Senior Principal Engineer at Amazon. He wrote a great [article](#) for the AWS Builder’s Library on the challenges he faced while building distributed systems during his 20 years at Amazon.

Here’s a summary

Types of Distributed Systems

Distributed systems can be divided into different categories, and some categories have more challenges than others.

On the “easier” side (but still far from trivial to implement) are **Offline Distributed Systems** where you take a [batch job](#) and split it up across many machines that are located in close proximity. These systems are frequently used for big data analysis or [high performance computing](#). You can get almost all the benefits of distributed computing (scalability and fault tolerance) and avoid much of the downside (complex failure modes and non-determinism).

In the middle are **Soft Real-Time Distributed Systems**. These are systems that must continually produce or update results, but have a relatively generous time window in which to do so (hence soft real-time). Things like web crawlers, search indexers, ML training infrastructure, etc. The system can go down for several hours without undue customer impact.

The most difficult are **Hard Real-Time Distributed Systems**. These are request/reply services where clients will randomly send requests and expect an immediate reply. Web servers, credit card processors, every AWS API, etc. are examples

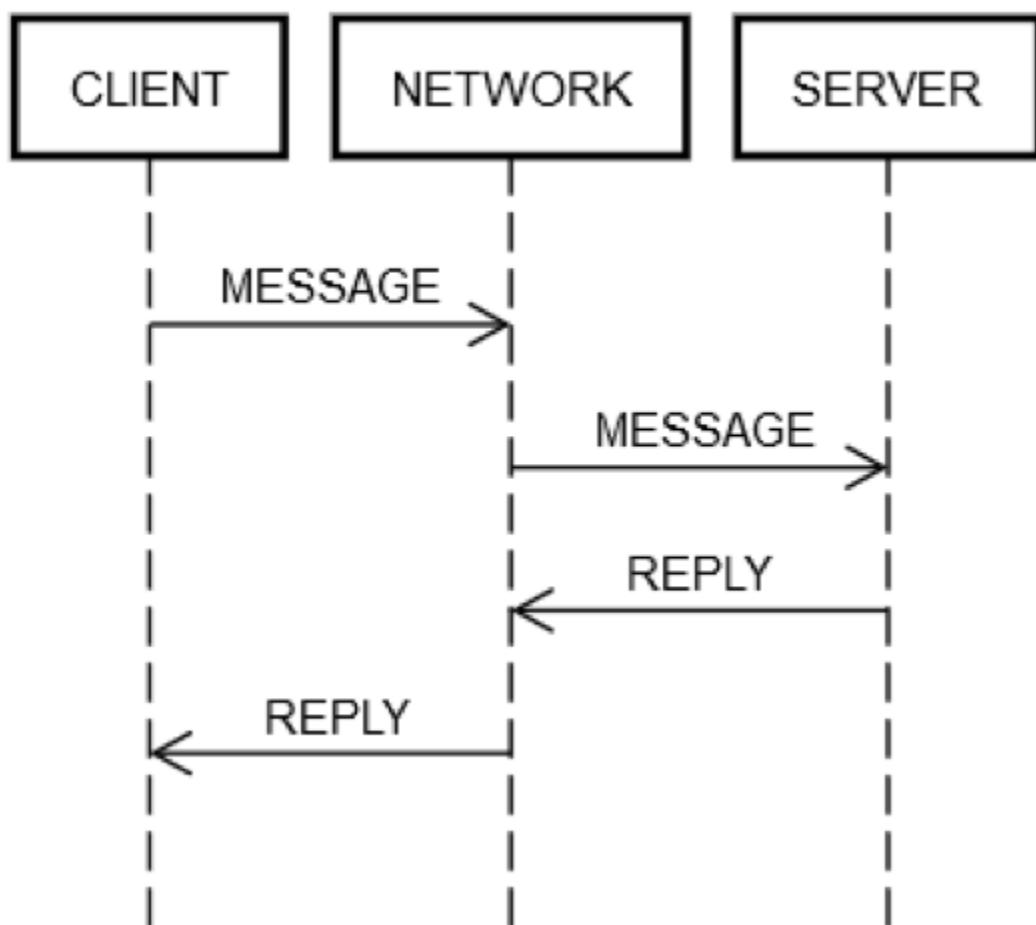
of hard, real-time distributed systems. The article delves into why these systems are difficult to build.

Complexity

Request/reply networking is the main reason why hard, real-time distributed systems are so challenging. Regardless of what protocols you're using, using the network means you're sending messages from one fault domain to another.

This introduces many steps where something can go wrong. As your systems grow larger, what had previously been theoretical edge cases will turn into regular occurrences due to the law of large numbers.

Here are the steps involved with request/reply networking.



1. Post Request - The client sends the request message onto the network.
2. Deliver Request - The network delivers the message to the server.
3. Validate Request - The server validates the message.
4. Update Server State - The server may update its state based on the message.
5. Post Reply - The server sends a reply onto the network.
6. Deliver Reply - The network delivers the reply to the client.
7. Validate Reply - The client validates the reply.
8. Update Client State - The client may update its state based on the reply.

Creating a distributed system means introducing all of these steps into your program. It turns one step (calling a method or writing to disk) into eight steps that will each fail with some non-zero probability.

Handling Failure Modes and Testing

When you're working with a single machine, [fate sharing](#) reduces the complexity of the testing process. Fate sharing is where when one component of the system fails, then everything else will fail too. It cuts down on the different failure modes that you have to handle.

With a single machine, you don't have to test for conditions where the CPU dies. If the CPU dies on your laptop, then those test conditions obviously won't be processed anyway.

However, in hard real-time distributed systems, the client, network and server do not share fate. One of the machines can die on the backend but the other machines, the client and the network will still function as normal.

This means testing for all possible failure scenarios and controlling for code behavior during these faults. The increased number of failure modes multiply the number of test conditions.

Previously, you had to write a test for handling bugs in the method you were calling. Now, you still need those tests but you also need to test for network failures, unrelated server failures, delayed responses, no responses, etc.

Each of those eight steps in request/reply networking introduce possible failure modes, and building distributed systems at scale means you have to test for all of them and handle all the permutations.

Distributed Bugs are Often Latent

If a failure is going to happen, common wisdom is that it's better if it happens sooner rather than later.

Distributed bugs (those that result from failing to handle all the permutations of the eight failure modes) are usually severe and can be caused by bugs that were deployed to production months earlier.

It takes a while to trigger the exact combination of scenarios that lead to these bugs happening, hence the delay.

Distributed Bugs Spread Epidemically

Another problem that is fundamental to distributed bugs is that they involve use of the network. Therefore, these bugs are more likely to spread and start to cause problems in other machines on the network.

This is especially true since distributed systems will have multiple layers of abstraction. Your system usually won't just be a single client, a network and a single server machine.

Instead, the backend will consist of multiple machines grouped together across different geographic regions.

Jason elaborates on this by giving a story of a bug that took down the Amazon website. It was caused by a single server failing within the remote catalog service when its disk filled up.

The failure was caused by a single server failing within the remote catalog service when its disk filled up. Due to mishandling of that error condition, the remote catalog server started returning empty responses to every request it received. It also started returning them very quickly, because it's a lot faster to return nothing than something (at least it was in this case). Meanwhile, the load balancer between the website and the remote catalog service didn't notice that all the responses were zero-length. But, it did notice that they were blazingly faster than all the other remote catalog servers. So, it sent a huge amount of the traffic from www.amazon.com to the one remote catalog server whose disk was full. Effectively, the entire website went down because one remote server couldn't display any product information."

For more details, you can read the full blog post [here](#).

How PayPal solved their Thundering Herd Problem

Braintree is a fintech company that makes it easy for companies to process payments from their customers. They provide a [payment gateway](#) so companies can process credit and debit card transactions by calling the Braintree API. In 2018, Braintree processed over 6 billion transactions and their customers include Airbnb, GitHub, Dropbox, OpenTable and more.

PayPal acquired Braintree in 2013, so the company comes under the PayPal umbrella.

One of the APIs Braintree provides is the [Disputes API](#), which merchants can use to manage credit card [chargebacks](#) (when a customer tries to reverse a credit card transaction due to fraud, poor experience, etc).

The traffic to this API is highly irregular and difficult to predict, so Braintree uses autoscaling and asynchronous processing where feasible.

One of the issues Braintree engineers dealt with was the [thundering herd problem](#) where a huge number of Disputes jobs were getting queued in parallel and bringing down the downstream service.

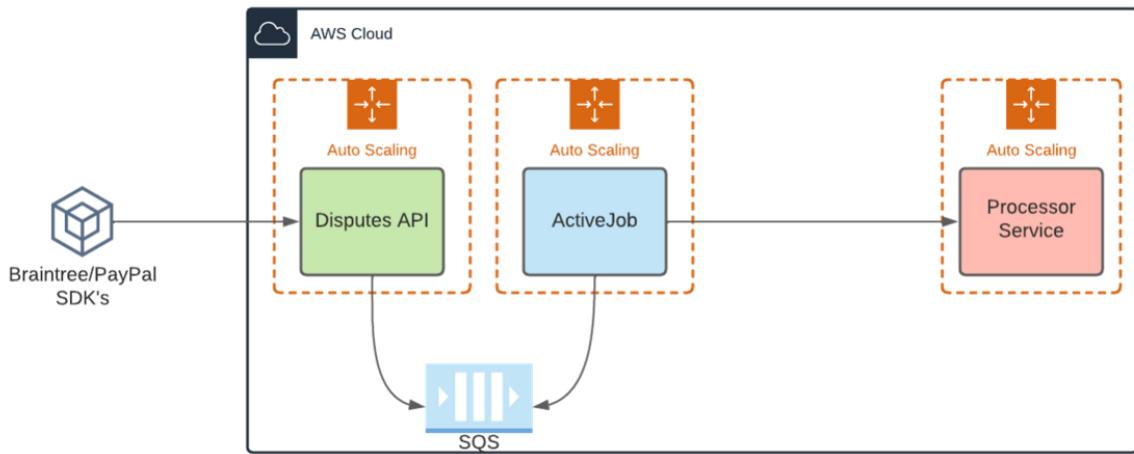
Anthony Ross is a senior engineering manager at Braintree, and he wrote a great [blog post](#) on the cause of the issue and how his team solved it with exponential backoff and by introducing randomness/jitter.

Here's a summary

Braintree uses [Ruby on Rails](#) for their backend and they make heavy use of a component of Rails called [ActiveJob](#). ActiveJob is a framework to create jobs and run them on a variety of queueing backends (you can use popular Ruby job frameworks like [Sidekiq](#), [Shoryuken](#) and [more](#) as your backend).

This makes picking between queueing backends more of an operational concern, and allows you to switch between backends without having to rewrite your jobs.

Here's the architecture of the Disputes API service.



Merchants interact via SDKs with the Disputes API. Once submitted, Braintree enqueues a job to AWS Simple Queue Service to be processed.

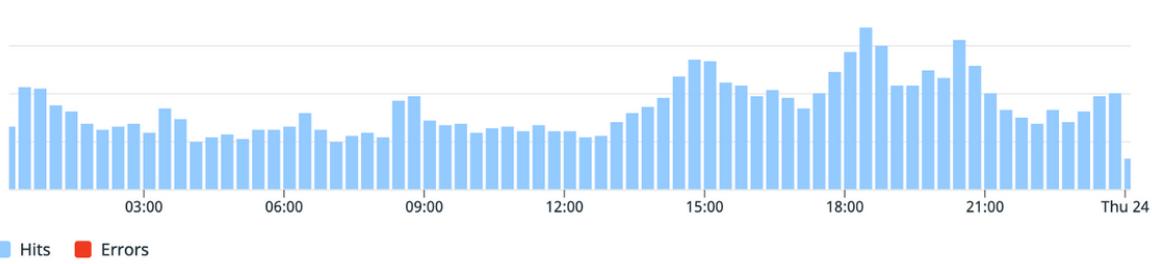
ActiveJob then manages the jobs in SQS and handles their execution by talking to various Processor services in Braintree's backend.

The Problem

Braintree set up the Disputes API, ActiveJob and the Processor services to autoscale whenever there was an increase in traffic.

Despite this, engineers were seeing a spike in failures in ActiveJob whenever traffic went up. They have a robust retry logic setup so that jobs that fail will be retried a certain number of times before they're pushed into the **dead letter queue** (to store messages that failed so engineers can debug them later).

The retry logic had ActiveJob attempt the retries again after a set time interval, but the retries were failing again.



The issue was a classic example of the [thundering herd problem](#). As traffic increased (and ActiveJob hadn't autoscaled up yet), a large number of jobs would get queued in parallel. They would then hit ActiveJob and trample down the Processor services (resulting in the failures).

Then, these failed jobs would retry on a static interval, where they'd also be combined with new jobs from the increasing traffic, and they would trample the service down again. The original jobs would have to be retried as well as new jobs that failed.

This created a cycle that kept repeating until the retries were exhausted and eventually DLQ'd (placed in the [dead letter queue](#)).

To solve this, Braintree used a combination of two tactics: Exponential Backoff and Jitter.

Exponential Backoff

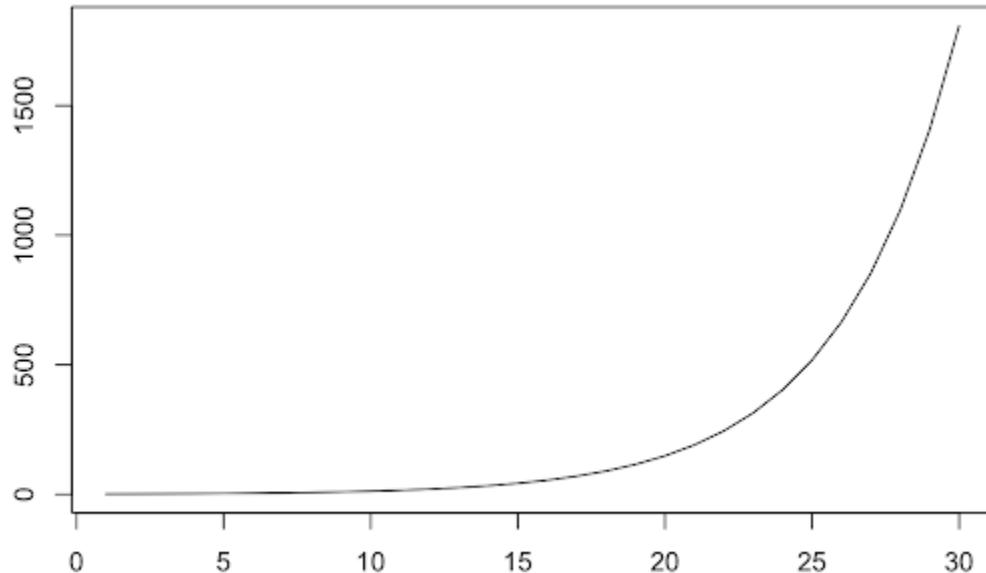
Exponential Backoff is an algorithm where you reduce the rate of requests exponentially by increasing the amount of time delay between the requests.

The equation you use to calculate the time delay looks something like this...

$$\text{time delay between requests} = (\text{base})^{\text{(number of requests)}}$$

where *base* is a parameter you choose.

With this, the amount of time between requests increases exponentially as the number of requests increases.



However, exponential backoff alone wasn't solving Braintree's problems.

By just using exponential backoff, the retries + new jobs still weren't spread out enough and there were clusters of jobs that all got the same sleep time interval. Once that time interval passed, these failed jobs all flooded back in and trampled over the service again.

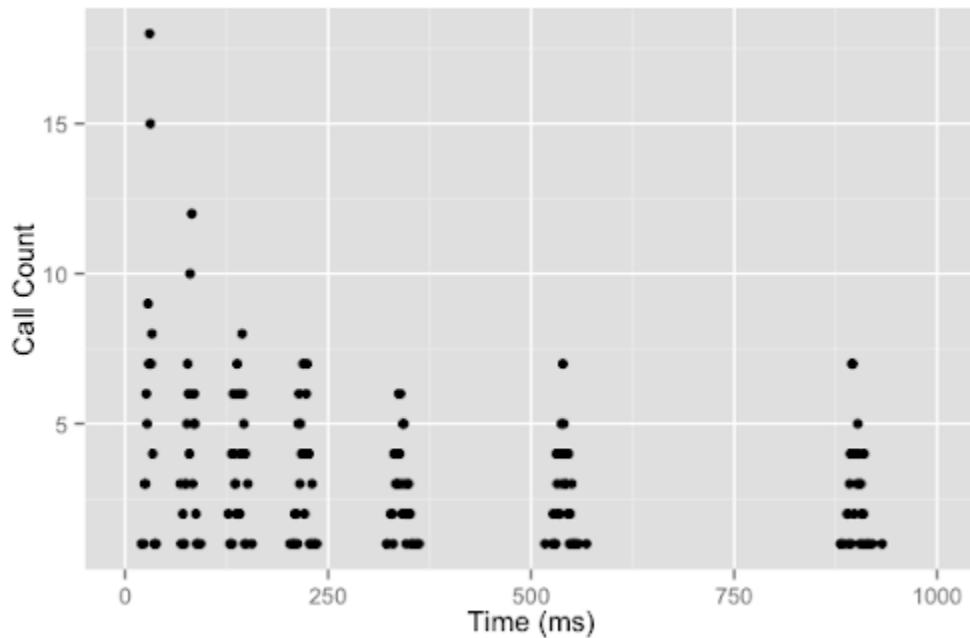
To fix this, Braintree added jitter (randomness).

Jitter

Jitter is where you add randomness to the time interval the requests that you're applying exponential backoff to.

To prevent the requests from flooding back in at the same time, you'll spread them out based on the randomness factor in addition to the exponential function. By adding jitter, you can space out the spike of jobs to an approximately constant rate between now and the exponential backoff time.

Here's an example of calls that are spaced out by just using exponential backoff.



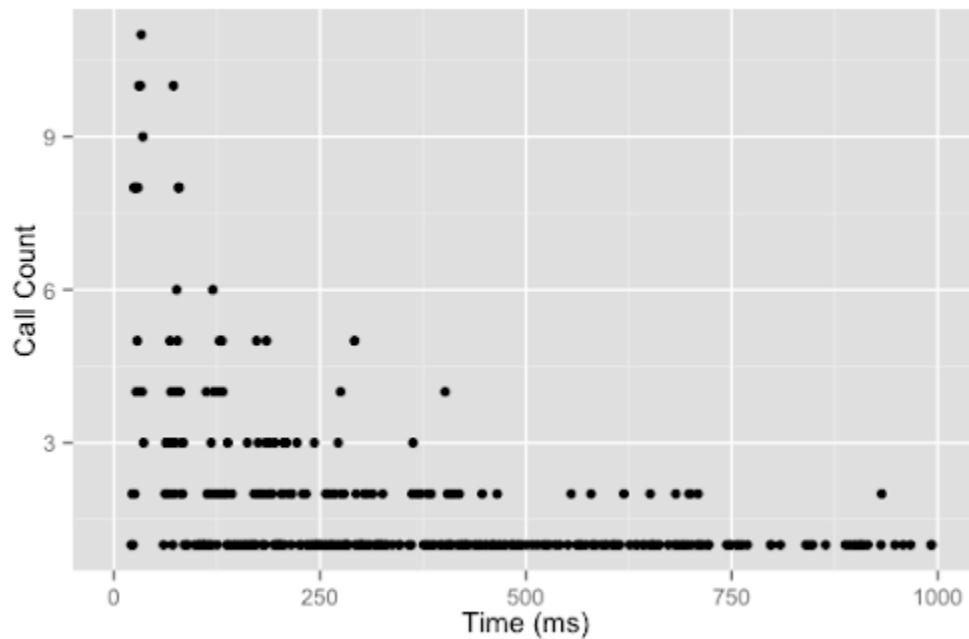
The time interval between calls is increasing exponentially, but there are still clusters of calls between 0 ms and 250 ms, near 500 ms, and then again near 900 ms.

In order to smooth these clusters out, you can introduce randomness/jitter to the time interval.

With Jitter, our time delay function looks like

time delay between requests = random_between(0, (base)^number of requests))

This results in a time delay graph that looks something like below.



Now, the calls are much more evenly spaced out and there's an approximately constant rate of calls.

For more details, you can read the full article by Braintree [here](#).

[Here's](#) a good article on Exponential Backoff and Jitter from the AWS Builders Library, if you'd like to learn more about that.

How Twitch Processes Millions of Video Streams

Twitch is a live streaming platform where content creators can stream live video to an audience. There are millions of broadcasters on the platform and tens of millions of daily active users. At any given time, millions of people are streaming video through the Twitch platform (on web, mobile, smart TV, etc.)

Twitch's Video Ingest team is responsible for developing the distributed systems and services that

- Acquire live streams from Twitch content creators
- Perform real-time processing ([transcoding](#), [compression](#), etc.)
- Provide a high throughput [control plane](#) to make the video available for world-wide distribution with low latency

Eric Kwong, Kevin Pan, Christopher Lafata and Rohit Puri are software engineers on the Video Ingest team and they wrote a great [blog post](#) on their infrastructure/architecture, problems they encountered and solutions they employed.

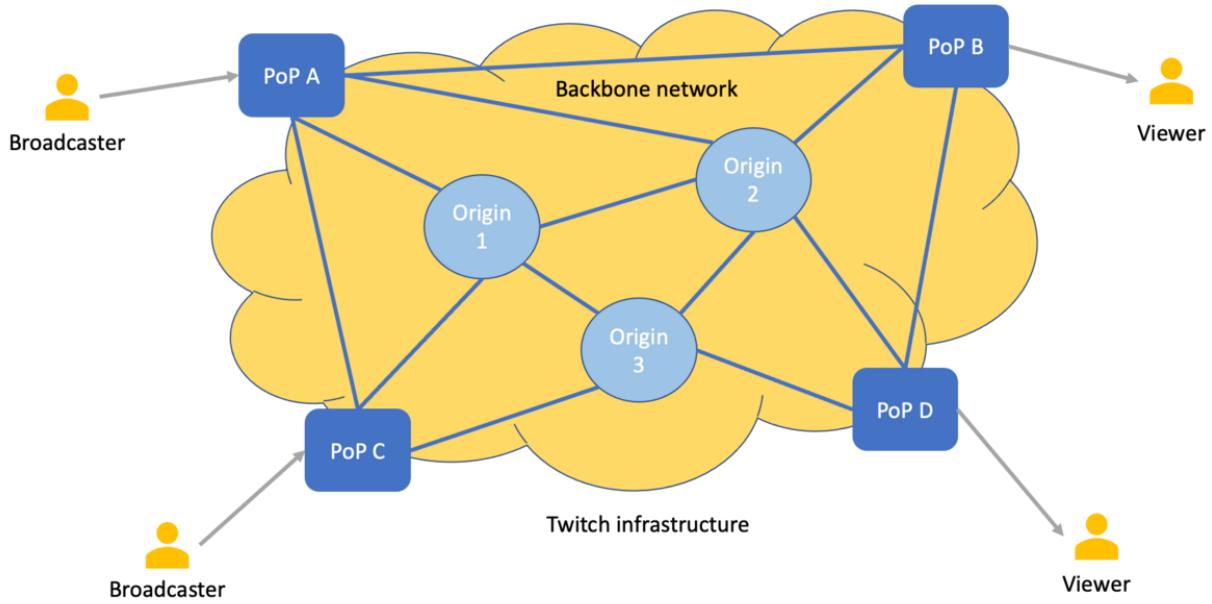
Here's a summary

Twitch maintains nearly a hundred servers (Points of Presence or PoPs) in different geographic regions around the world that streamers and viewers can connect to for uploading/downloading video.

These Points of Presence (PoPs) are connected through Twitch's private [Backbone Network](#), which is dedicated to transmitting their content. Relying on the public Internet would be susceptible to bottlenecks/instability so instead, 98% of all Twitch traffic remains on their private network.

Between the PoPs are origin data centers, which are also geographically distributed. These origin data centers handle tasks around video processing (like transcoding a livestream into different [bitrates](#)/formats for all the various devices that viewers may be using).

Video travels from a streamer's computer to a PoP. From there, it's sent to an origin data center for processing and then transmitted to all the PoPs that are close to the stream's viewers. This is all done over Twitch's Backbone network.



Previously, all the PoPs ran **HAProxy** (a reverse proxy that is commonly used for load balancing) for forwarding the video streams to the origin data centers. However, Twitch faced several issues with this approach as they scaled.

Inefficient Usage of Origin Data Center Resources - Each PoP was configured to send its video streams to a specific origin data center (located in the same geographic area as the PoP). This meant that the origin data centers for a region ran at full load during the busy hours of that geographic area, but utilization became very minimal outside of that time period. When one region has minimal utilization, another geographic region might be having their busy hours but they couldn't take advantage of the origin data centers of the minimal utilization region.

Difficult to Handle Unexpected Changes - The relatively static nature of the HAProxy configuration also made it difficult to handle unexpected surges of live video traffic. Reacting to system fluctuations like the loss of capacity of an origin data center was also very difficult.

Creating Intelligest

Twitch decided to revamp the software in their PoPs and completely retire HAProxy.

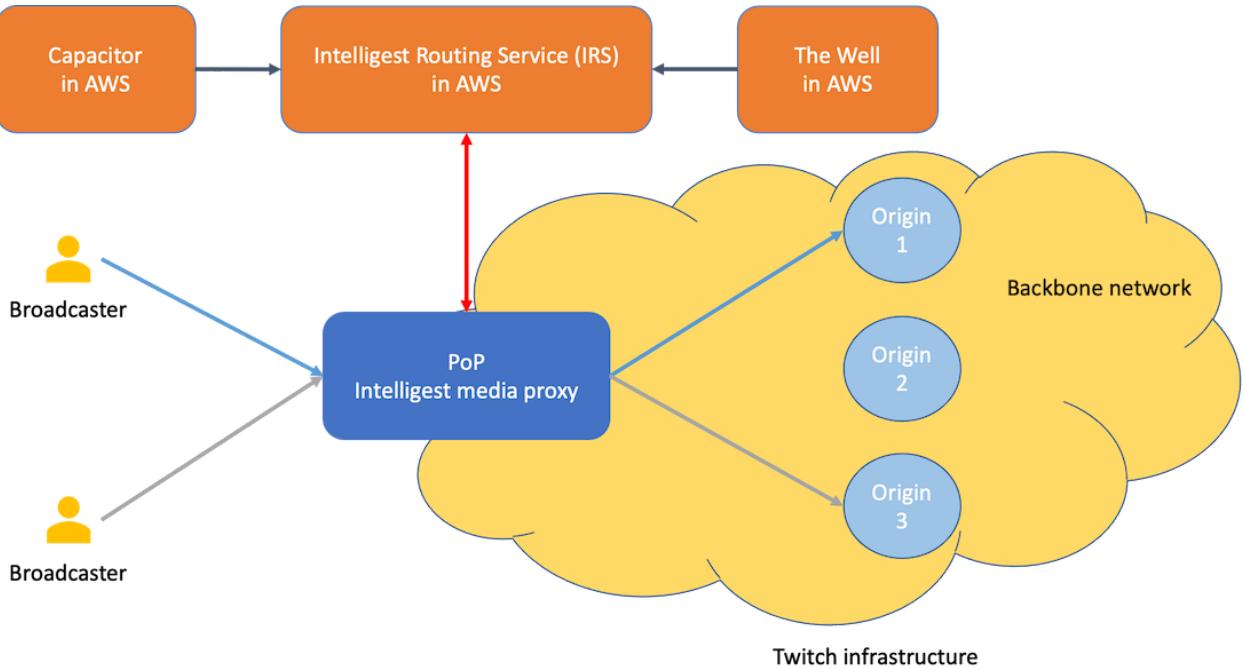
To replace it, they developed Intelligest, a proprietary ingest routing system that could intelligently distribute live video ingest traffic from the PoPs to the origins.

The Ingest architecture consists of two components: the Intelligest Media Proxy and the Intelligest Routing Service (IRS). The Intelligest Media Proxy is a **data plane** component so it runs in all the PoPs and sends the video streams to various origin data centers. The Intelligest Routing Service is a **control plane** and tells the Intelligest Media Proxy which origin data center to send the video to.

When a broadcaster starts streaming, his computer will transmit video to the nearest Twitch Point of Presence (PoP) server. The Intelligest Media Proxy is running on that PoP and it will extract all the relevant metadata from the stream.

It will then query the Intelligest Routing Service (IRS) and ask which origin data center it should route the video stream to. The IRS service has a real-time view of all of Twitch's infrastructure and it will make a routing decision based on minimizing latency for the viewers and maximizing utilization of compute resources in all the origins.

The IRS service will send its decision back to the Intelligest Media Proxy, which can then route the video stream to the selected origin data center.



The Intelligent Routing Service relies on two other services implemented in AWS: Capacitor and The Well.

Capacitor monitors the compute resources in every origin and keeps track of any capacity fluctuations (due to maintenance/failures).

The Well monitors the backbone network and provides information about the status of network links so latency issues are minimized.

The IRS service uses a [randomized greedy algorithm](#) to compute routing decisions based on compute resources available, backbone network bandwidth and other factors.

For more details, you can read the full blog post [here](#).

How Image Search works at Dropbox

Dropbox is a file hosting and sharing company with over 700 million registered users.

Photos are among the most common types of files uploaded to Dropbox. The Dropbox app allows users to set up camera sync so that any photo they take on their smart phone will automatically get synced and stored in their Dropbox account.

To make it easier for people to find their photos, Dropbox built an image search feature where you can search for objects/scenery/action and Dropbox will find images that contain what you searched for.

For example, if you search for “picnic”, then Dropbox will find your images that contain a picnic.

The screenshot shows a search interface for 'photos'. The search bar contains 'photos' and 'picnic'. Below the search bar, it says '2 results' and 'Reset filters'. There are dropdown menus for 'photos' and 'By type', and a 'Relevance' dropdown. Two thumbnail images are displayed: one of a man sitting at a picnic table with food, and another of a man and a woman sitting on grass eating.

photos x By type x Relevance x

2 results Reset filters

photos v By type v

Relevance v

2017-07-04 12.37.54.jpg P7250003.jpg

Thomas Berg is a Machine Learning Engineer at Dropbox and he wrote a great [blog post](#) summarizing how this feature works.

Here's a summary

Dropbox's Approach

To build this, Dropbox relies on two areas of machine learning

- Image Classification
- Word Vectors

Image Classification

An image classifier takes in the pixel values of an image and outputs a list of things that the image contains (where each of these things are categories that the classifier is trained to recognize).

There has been tremendous progress over the last 10 years in image classification with the innovations in deep learning, specifically [convolutional neural networks](#). Model architecture improvements, better training methods, large datasets, faster GPUs and more have resulted in image classifiers that can recognize thousands of different categories with extremely high accuracy.

For Dropbox's image search, their image classifier is an [EfficientNet network](#) trained on the [OpenImages dataset](#). This produces classification scores for ~8500 categories ranging from grapes to telephone to picnic and much more.

However, an issue that comes up with image classification is synonyms. What if a user searches for seashore but the image classifier is trained on the term beach?

To solve this, Dropbox used [word vectors](#).

Word Vectors

Word vectors are an extremely important technique in natural language processing that really took off in 2013 with [Word2vec](#).

The idea is that you have a [vector space](#) with hundreds of dimensions (your standard X-Y cartesian coordinate system could be viewed as an example of a 2 dimensional vector space).

Then, you use a neural network to map every word to a vector in the vector space. For each word, the neural network will assign a number for each of the hundreds of different dimensions.

You train the neural network so that words with similar meanings will be assigned vectors that are close to each other in vector space. [Here's](#) a great article that dives deeper into Word2vec.

Dropbox used the [ConceptNet Numberbatch](#) pre-computed word embeddings. This gave them good results in their testing and the word embeddings also support multiple languages, so they return close vectors for words in different languages with similar meanings. This makes supporting image search in multiple languages much easier.

If there's a multi-word image search, Dropbox parses the query as an AND of the individual words. They also maintain a list of multi-word terms like beach ball that they can match for.

Production Architecture

When a user submits a search query, it's obviously not possible to immediately run image classification on all of their images. Users can have tens of thousands of images in their dropbox account, so that solution would be way too slow.

Instead, Dropbox uses an [Inverted Index](#) data structure (also used by many Full-Text search engines like Elasticsearch). You can think of an Inverted Index as very similar to the Index section at the back of the textbook where it contains a list of all the words in

the book and the corresponding page numbers where each word is mentioned. An Inverted Index contains a mapping from all the unique words/phrases in the text to their locations in the documents.

Dropbox will scan through all the images in a user's account and run their image classification algorithm to find all the categories (things) that appear in that image. They convert whatever categories are found into the corresponding word embedding vectors.

Then, they create an Inverted Index where for each category, they have a list of images that contain that thing.

Forward index

```
image_1: filename: "2019-06-06 14.35.55.jpg", cat_vec: (0.92, 0.00, 0.00, 0.98, 0.99, 0.10, ...)  
image_2: filename: "2019-06-06 15.20.01.jpg", cat_vec: (0.01, 0.99, 0.00, 0.92, 0.01, 0.82, ...)  
image_3: filename: "letterhead-logo.png", cat_vec: (0.00, 0.01, 0.97, 0.0, 0.0, 0.01, ...)  
...
```

Inverted index

```
blanket: image_1, image_2  
dog: image_2, image_3  
logo: image_3  
park: image_1, image_2  
sandwich: image_1, image_2  
tree: image_1, image_2, image_3  
...
```

Search index contents for image content search

When a user searches for a word, Dropbox will first find the word vector for that term.

Then, they'll find the closest word vectors that are categories for the image classifier.

They'll query the inverted index to find the matching images for these categories and then rank the matching images based on how strong the classifier ranked each category in the image.

For more details and some additional optimizations Dropbox made, you can view the full article [here](#).

How Instagram Suggests New Content

Instagram is a social media app with more than 2 billion active users. In order to keep users engaged, the company dedicates a ton of resources to making sure the posts in a user's feed are relevant, fresh and interesting.

Instagram launched a Suggested Posts feature where they recommend posts a user may enjoy from accounts the user isn't following and they place those posts in the user's feed. The goal is to make it easier for users to find new accounts to follow.

Amogh Mahapatra is a machine learning engineer at Meta and he wrote a great [blog post](#) on how Instagram implemented this feature.

Here's a summary

Instagram's suggested posts feature will find photos/video posts that you may like from accounts that you don't follow. This results in you finding more content you like, following more accounts and spending more time on Instagram.

This feature is an example of the [Information Retrieval](#) problem, where you have a large set of documents (Instagram posts) and you want to find certain documents based on a set of criteria.

Information Retrieval systems typically have a two-step design

1. [Candidate Generation](#) - based on the user's interests, fetch all the candidates that a user could be interested in. In this case, Instagram is looking for all the possible posts from accounts the user doesn't follow that he/she may be interested in.
2. [Candidate Selection/Scoring](#) - rank the candidates and select the best subset to show to the user. In this scenario that means looking at the potential posts from the candidate generation stage and selecting the best few posts that will be shown to the user as Suggested Posts in their feed.

How Instagram does Candidate Generation

The first step is to search for posts that a user may like from accounts the user isn't following. To do this, Instagram relies on user embeddings and co-occurrence based similarity.

User embeddings are a popular technique in building recommendation systems, where you use a machine learning model to generate a [vector](#) that represents a user. The vector is a series of numbers (magnitudes) in various dimensions. These numbers are chosen by the ML model based on the user's engagement data so users who have similar engagement data will get similar numbers. You can find the most similar accounts to the user by looking at other accounts that are nearby in vector space (using something like [Cosine similarity](#)).

This is based on [word embeddings](#), where you generate a vector representation of a word based on the meaning and usage of that word. We talked about word embeddings in a previous article on [how Dropbox implemented their image search feature](#).

Instagram also uses a technique called Co-occurrence Similarity, which comes from [frequent pattern mining](#). They look at user data to see what media users are engaging with and look for any co-occurring accounts (accounts that also get engagement from those users). Then, they calculate co-occurrence frequencies of media pairs and use them for Candidate Generation. For example, there may be a lot of users who like posts from the Golden State Warriors and also from the Los Angeles Lakers (two NBA teams). Users who follow one team and not the other might benefit from getting the other team's posts as Suggested Posts.

Cold Start Problem

An issue that you'll frequently see with recommender systems is the **Cold Start problem**, where the system performs poorly for new users due to a lack of data.

Instagram deals with this in two ways:

- Popular Media - For extremely new users who don't follow anyone / haven't engaged with any content, Instagram will recommend posts that are popular with the general Instagram user base. The recommendation algorithm can then adjust based on the user's response to those initial posts.
- Fallback Graph Exploration - If a user hasn't engaged with any content but follows other accounts, Instagram will generate candidates for them by evaluating their one-hop and two-hop connections. They'll look at accounts followed by the user and see what posts those accounts liked and use that to generate candidates.

How Instagram does Candidate Selection

The candidate generation step generates a group of potential Suggested Posts. In the candidate selection stage, machine learning models are used to pick the best few posts that'll be shown to the user.

To do this, Instagram uses a ton of different data points and various machine learning models. Many of the data points are also generated using ML models.

Some of the data points considered are

- Probability of user engagement
- Content quality of the image/video
- Past Author-Viewer interactions/engagement
- User embeddings

And much more.

Some of the models used are

- Log-linear models
- Gradient Boosted Decision Trees
- Multi Task Multi label Sparse Neural Nets (MTML)

In order to select the best models, hyperparameters, etc. Facebook relies on online A/B testing and offline simulations. The offline simulations work by replaying a user's actions (their likes, comments, shares, etc.) to different models and training them to predict the user's actions. Then, these engagement prediction models can be used to evaluate candidate ranking models.

Offline simulation can't replace A/B testing since there are many behavioral dynamics that are too complicated to model, but it provides a higher throughput alternative to quickly evaluate model performance. You can read more about offline simulation at Meta [here](#).

For more details on Instagram's Suggested Post feature, read the full article [here](#).

How Snapchat Works

Snapchat is an instant messaging app with over 300 million daily active users. The company uses a multi-cloud strategy relying heavily on AWS and Google Cloud Platform.

Saral Jain is the Senior Director of Engineering at Snap Inc where he leads the Cloud Infrastructure, Data and IT organizations. He gave a [great interview](#) on the AWS series This is my Architecture.

He discussed the process of what happens on the backend when a user sends/receives a snap on the app (sends or receives an image/video). This is for a video series by AWS, so unfortunately he only talks about the AWS architecture.

Here's a summary

For their AWS stack, Snap runs their backend on Elastic Kubernetes Service (EKS) and they use more than 900 EKS clusters where many of the clusters have 1000+ instances.

The core services involved in sending and receiving snaps are the

- Media Delivery Service
- Core Orchestration Service
- Friend Graph
- Snap DB



When a user sends a snap from their mobile device, their phone will talk to Snapchat's API Gateway.

The Gateway will communicate with the Media Delivery Service to send the picture/video to AWS CloudFront (AWS' Content Delivery Network) and also persist it in S3. The media will be given a media ID that it can be referenced through.

Once the media has been persisted, Snap's Orchestration service will query Snapchat's friend graph to make sure that the sender has the permissions to send the picture/video to the recipient (they should be friends on Snapchat).

If the permissions check passes, the Orchestration service will persist the conversation metadata (including the media ID) into Snap DB. Snap DB is Snapchat's custom database that is built on top of DynamoDB (a proprietary NoSQL database by AWS). They store nearly 400 terabytes of data in DynamoDB.

The team created their own database as a frontend to DynamoDB to add higher level features to meet Snap's specific use cases. Snap has to deal with a lot of ephemeral data so they added optimizations for that and also TTL and custom transactions to reduce costs.

For receiving a snap, the orchestration service will look up a connection ID from ElasticCache, to get access to the persistent connection that Snap servers have with the clients who have the app open.

The service looks at the conversation metadata to get the media ID of the picture/video. The content is retrieved from CloudFront and then sent to the recipient's device.

If the recipient doesn't have the app open, then Snapchat relies on Apple Push Notification Service or Firebase Cloud Messaging.

For more details, you can watch the full video [here](#).

How Netflix Implemented Load Shedding

The API Gateway sits between the backend and the client and it handles things like rate limiting, authentication, monitoring and routing requests to all the various backend services.

There are a ton of different tools you can use for an API Gateway like Nginx, Zuul (by Netflix), Envoy (by Lyft), offerings from all the major cloud providers, etc.

One feature that many API gateways provide is load shedding, where you can configure the gateway to automatically drop certain requests and ignore them. This is crucial for times when you face a spike in traffic or if something's wrong with your backend (and you can't handle the usual traffic).

Non-critical requests like logging or background requests can be dropped/shed by the API gateway so that critical requests (that impact the user experience) have fewer failures.

Netflix built and maintains a popular API Gateway called [Zuul](#) and they gave a [great talk](#) at AWS Re:Invent 2021 about how they designed and tested Zuul's prioritized load shedding feature for their internal use.

Here's a summary

Despite all the effort Netflix engineers put into developing resiliency, there are still many different incidents that degrade user experience.

Whether it's something like under-scaled services, network blips, cloud provider outages, bugs in code or something else, engineers need to ensure that the end user experience is minimally affected. Netflix is a movie/tv-show streaming website, so this means that users should still be able to stream their movies and TV shows on their phone/laptop/TV/gaming console.

To ensure high availability, Netflix uses the load-shedding technique where low priority requests are dropped when the system is under severe strain.

An example of a request that can be dropped is requests for show trailers. When a user is scrolling through Netflix, the website will autoplay the trailer of whatever movie/TV show the user currently has selected.

If the system is under severe strain, then Netflix will ignore these requests and the client will fall-back on just displaying the show's image and not playing any trailer. This doesn't result in a severe degradation in user experience and it allows the system to prioritize requests that directly relate to the streaming experience for users.

In order to implement prioritized load shedding, Netflix engineers went through 3 steps

1. Define a Request Taxonomy - Create a way to categorize requests by priority and assign a score to each request that describes how critical the request is to the user streaming experience.
2. Implement the Load Shedding Algorithm - Netflix chose to implement it in their API Gateway, [Zuul](#)
3. Validate Assumptions using Fault Injection - The [Chaos Engineering](#) discipline started at Netflix and the company uses those principles for testing system resilience in a scientific way.

We'll go through each of these steps

Define a Request Taxonomy

Netflix created a scoring system from 0 to 100 that assigns a priority to a request, with 0 being the highest priority and 100 being the lowest priority.



The score was based on 4 dimensions

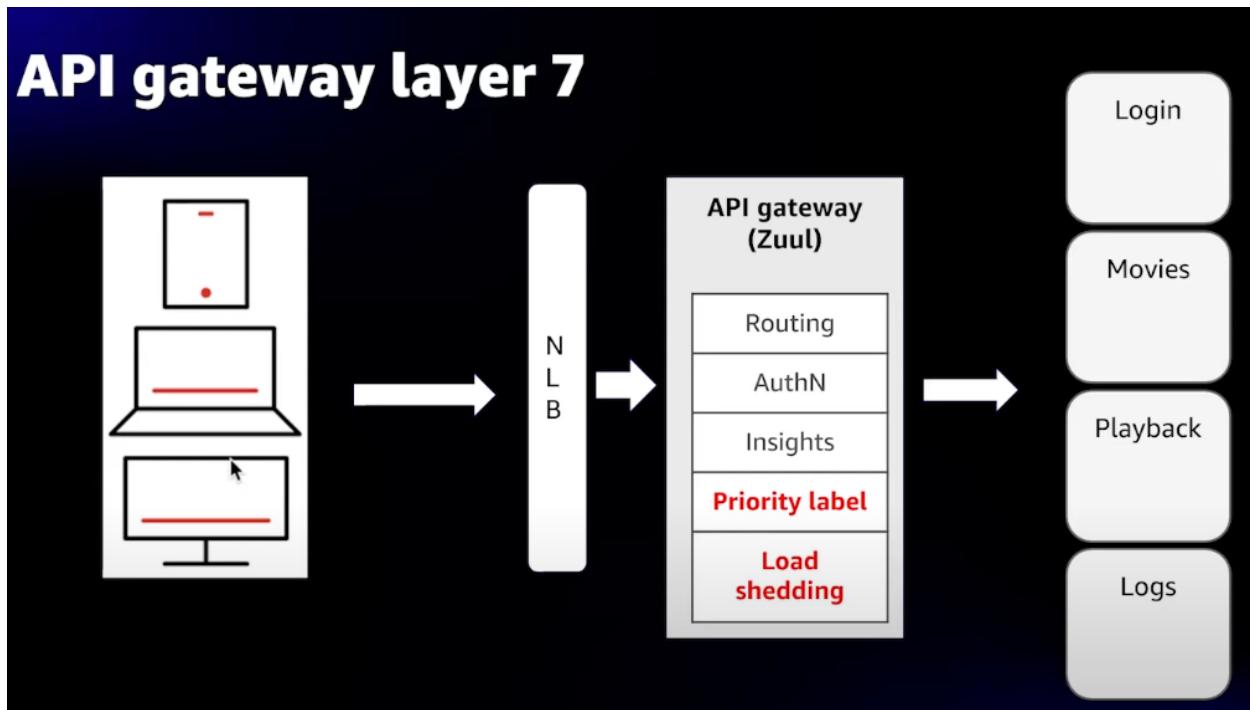
- Functionality - What functionality gets impacted if this request gets throttled? Is it important to the user experience? For example, if logging-related requests are throttled then that doesn't really hurt the user experience.
- Throughput - Some traffic is higher throughput than others. Logs, background requests, events data, etc. are higher throughput and they contribute to a large percentage of load on the system. Throttling them will have a bigger impact on reducing load.
- Criticality - If this request gets throttled, is there a well-defined fallback that still delivers an acceptable user experience? For example, if the client's request for the movie trailer gets blocked, then the fallback is to just show the image for the movie. This is acceptable.
- Request State - Was the request initiated by the user? Or was the request initiated by the Netflix app?

Using these dimensions, the API gateway assigns a priority score to every request that comes in.

Dimensions	Potential traffic drop	Priority
Low impact, high throughput	~40%	>= 90
Low impact, background	~60%	>= 80
Higher impact, lower throughput	~70%	>= 60
Higher impact, experience degradation	~80%	>= 40
High impact	~95%	>= 10
Full outage	100%	>= 1

Load Shedding Algorithm

The first decision was where to implement the load shedding algorithm. Netflix decided to put the logic in their API Gateway, Zuul.



(Note - NLB stands for [Network Load Balancer](#))

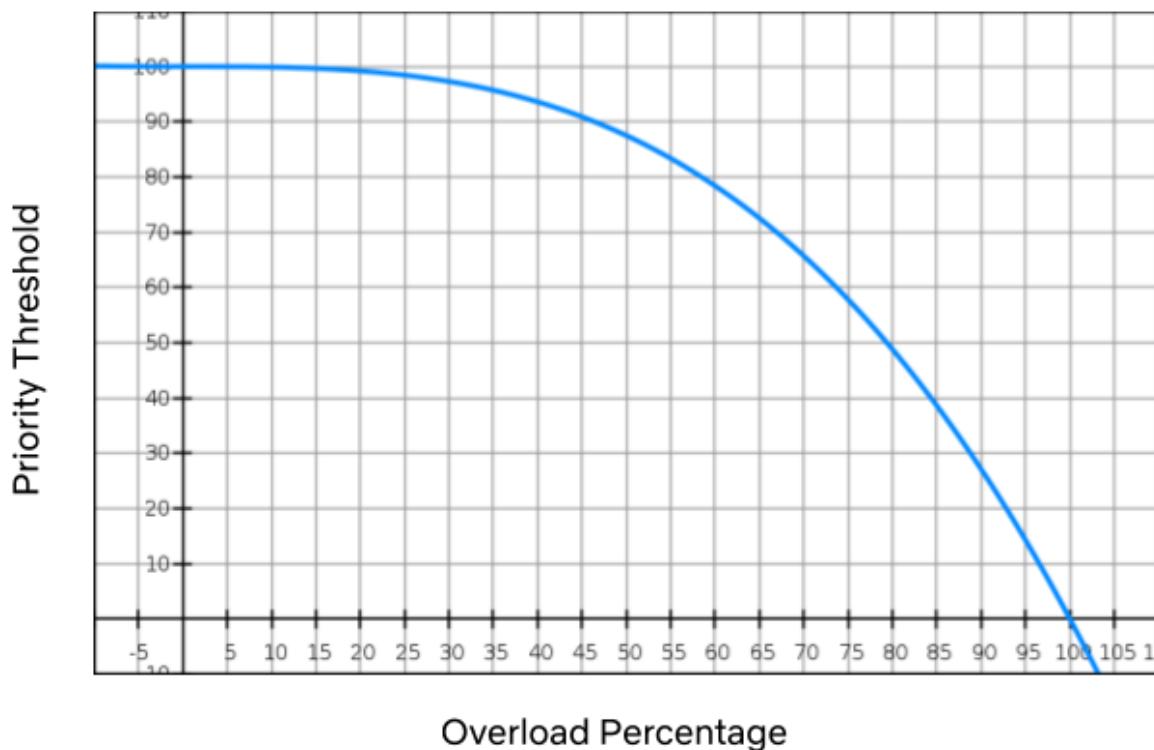
When a request comes in to Zuul, the first thing that the gateway does is execute a set of inbound filters. These filters are responsible for decorating the incoming request with extra information. This is where the priority score is computed and added to the request.

With the priority score information, Zuul can now do global throttling. This is where Zuul will throttle requests below a certain priority threshold. This is meant to protect the API gateway itself. The metrics used to trigger global throttling are concurrent requests, connection count and CPU utilization.

Netflix also implemented service throttling, where they can load shed requests for specific microservices that Zuul is talking to. Zuul will monitor the error rate and concurrent requests for each of the microservices. If a threshold is crossed for those

metrics, then Zuul will reduce load on the service by throttling traffic above a certain priority level.

In order to calculate the priority level, Netflix uses a cubic function. When the overload percentage is at 35%, Netflix will shed any requests that are above 95% priority. When the overload percentage reaches 80%, then the API Gateway will shed any request with a priority score of greater than ~50.



Validating Assumptions using Chaos Testing

A Fault Injection experiment is where you methodically introduce disruptive events (spike in traffic, CPU load, increased latency, etc.) in your testing or production environments and observe how the system responds.

Netflix routinely runs these types of experiments in their production environment and have built tools like Chaos Monkey and ChAP (Chaos Automation Platform) to make this testing easier.

They created a failure injection point in Zuul that allowed them to shed any request based on a configured priority. Therefore, they could manually simulate a load shedded experience and get an idea of exactly how shedding certain requests affected the user.

Engineers staged an A/B test that will allocate a small number of production users to either a control or treatment group for 45 minutes. During that time period, they'll throttle a range of priorities for the treatment group and measure the impact on playback experience.

This allows Netflix to quickly determine how the load shedding system is performing across a variety of client devices, client versions, locations, etc.

For more details, you can watch the full talk [here](#).

The Architecture of Facebook's Distributed Message Queue

Facebook uses thousands of distributed systems and microservices to power their ecosystem. In order to communicate with each other, these microservices rely on a [message queue](#).

Facebook Ordered Queueing Service (FOQS) is an internal Facebook tool that fills that role. FOQS is a horizontally scalable, persistent, distributed priority queue that's built on top of sharded MySQL.

Akshay Nanavati and Girish Joshi are two software engineers at Facebook, and they wrote a great [blog post](#) on how FOQS works and the architecture behind it.

Here's a Summary

FOQs Use Cases

FOQS is a general purpose priority queue so hundreds of different services across the Facebook stack rely on it to pass messages. Facebook's video encoding service, language translation technologies and notification services are a few examples.

Producer services will enqueue items on to FOQS to be processed. These items can have a priority and also a delay (if the item processing needs to be deferred). The item will have a topic, where each topic is a separate priority queue.

Consumer services can dequeue items from a certain topic and process them. If the processing succeeds, they send an "ack" message back to FOQS. If the processing fails, then they send a "nack" message back and the items will be redelivered from the priority queue at a later time.

Building a Distributed Priority Queue

FOQS is organized into namespaces where each namespace has many topics and each topic has many items.

Namespaces provide a way to separate all the different services/use-cases that are using FOQS. Each namespace will have many (thousands) of topics, where each topic represents a single priority queue.

Clients will enqueue and dequeue items to a topic where an item represents a message with some user specified data.

Each item will have fields for the namespace, topic, priority (a 32 bit integer), payload (an immutable 10 kilobyte blob), metadata, delivery delay (how long until the item can be dequeued) and a few other fields.

FOQS provides an API that consists of the following operations

- Enqueue - Add an item to FOQS.
- Dequeue - Accepts a topic and a number where the number signifies how many items to return from the topic. Items are returned based on priority and delivery delay.
- Ack - Sends a message that the dequeued item was successfully processed, so it doesn't need to be delivered again.
- Nack - Sends a message that the dequeued item needs to be redelivered because client processing failed. The processing can be deferred, allowing clients to leverage [exponential backoff](#) to give enough of a “cooling-off” period of buffer.
- GetActiveTopics - returns a list of the topics that have items

We'll go through how these operations work under the hood.

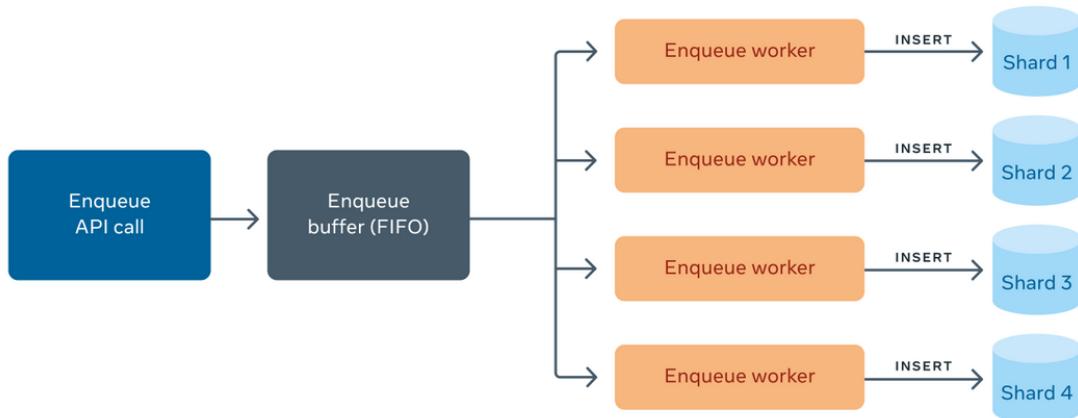
Enqueue

When a client enqueues an item to FOQS, the request gets put on an Enqueue Buffer and FOQS returns a promise back to the client.

FOQS is built on top of sharded MySQL and each shard has a corresponding worker node. The workers are reading items from the Enqueue Buffer and inserting them into their MySQL shard where one database row corresponds to one item.

Once the row insertion is complete, the promise is fulfilled and an enqueue response is sent back to the client. The response contains a unique string that contains the MySQL shard's ID and a 64-bit primary key (that identifies the item in its shard).

FOQS uses a circuit breaker design pattern to avoid sending items to unhealthy MySQL shards. Health is defined by slow queries or error rate; if either of those cross a threshold then the corresponding worker will stop accepting more work until it's healthy.



Dequeue

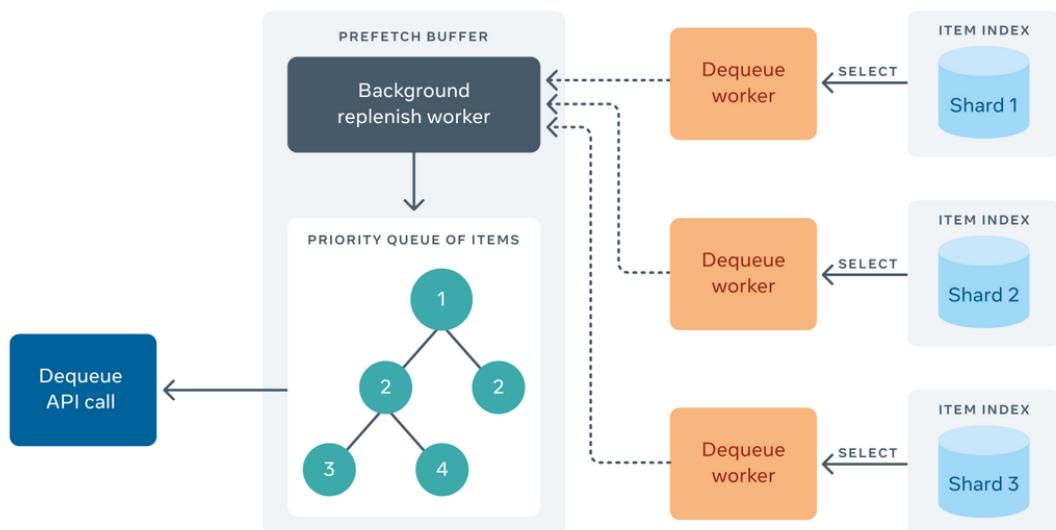
The dequeue API accepts a collection of (topic, count) pairs where *count* represents the number of items to return from the topic. The items returned are ordered by priority.

Since each topic is sharded, each topic host will need to run a *reduce* operation across all the MySQL shards for that topic to find the highest priority items and select those.

To optimize this, FOQS has a data structure called the Prefetch Buffer that works in the background and fetches the highest priority items across all the shards.

Each shard has an in-memory index of the primary keys of items that are ready to be delivered on the shard, sorted by priority. The Prefetch Buffer will build its own priority queue from these indexes using a [K-way merge](#).

The dequeue API just has to read items out of the Prefetch Buffer and return them to the client.



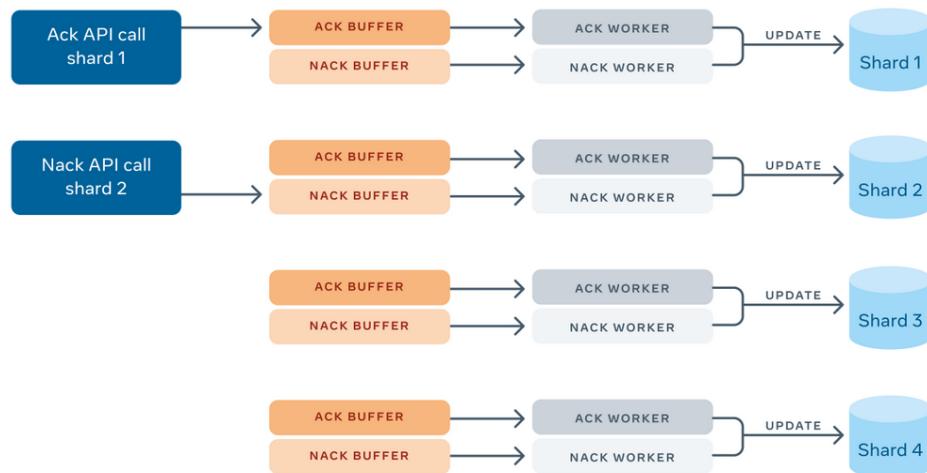
Ack/Nack

FOQS supports *at least once* delivery and that's implemented using Ack/Nack (short for Acknowledged or Not Acknowledged). An ack signifies that the dequeued item was successfully processed by the consumer, so the message doesn't need to be delivered again. A nack signifies that the item should be redelivered because the consumer client failed to process it.

When an item is enqueued, FOQS allows the client to specify a lease duration. When that item gets dequeued, the lease begins. If the item is not acked or nacked within the lease duration, it is assumed to have failed (nacked) and it's made available for redelivery, so that the *at least once* guarantee is met.

When an item succeeds/fails, the client sends the ack/nack request to FOQS. The shard ID is contained in the item ID, so the FOQS client uses that ID to locate the specific FOQS shard that manages that item.

The ack/nack gets sent to a shard-specific in-memory buffer, there are separate buffers for acks vs. nacks. A worker will pull items from the ack buffer and delete those rows from the MySQL shard. Similarly, a worker will pull items from the nack buffer and update that row with a new deliver_after time so the item gets redelivered.



For more details, you can read the full blog post [here](#).

How Mixpanel Fixed their Load Balancing Problem

Mixpanel is an analytics product that you can embed into your website/app to get detailed data on how your users are behaving (similar to Google Analytics).

In order to best serve their users, Mixpanel needs to support real-time event ingestion while also supporting fast analytical queries over all a user's history.

When a user on a Mixpanel-tracked website clicks a button or navigates to a new page, that event needs to be ingested and stored in Mixpanel in under a minute (*real-time event ingestion*).

If a Mixpanel customer wants to see the number of sign up conversion events over the past 6 months, they should be able to query that data quickly (fast analytical queries).

Mixpanel accomplishes this with their in-house database, Arb. They leverage both **row-oriented** and **column-oriented** data formats where row-oriented works better for real-time event ingestion and column-oriented works well for analytical queries. This is based on the classic **Lambda Architecture** where you have a speed layer for real-time views and a batch layer for historical data.

If you're interested in learning more about Mixpanel's system architecture, you can read about it [here](#).

In order to convert data from row format to a columnar format, Mixpanel has a service called Compacter.

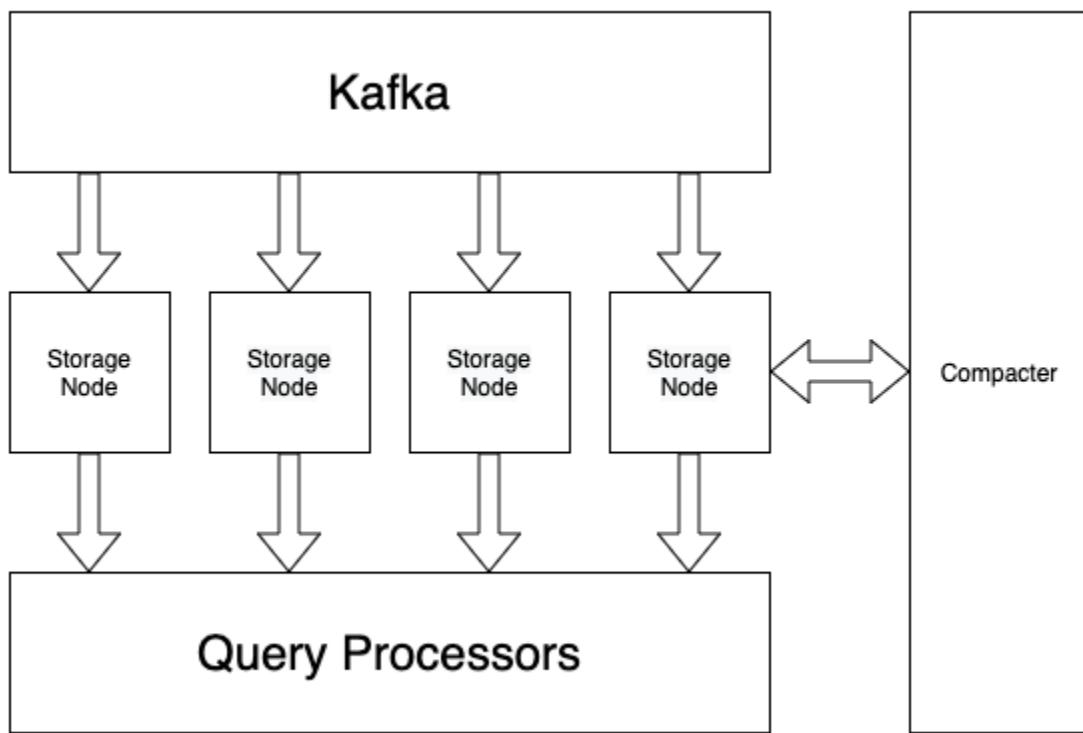
Vijay Jayaram was the Principal Tech Lead Manager of the Performance team at Mixpanel, and he wrote a great [blog post](#) on technical challenges the company faced when scaling the Compacter service and how they overcame them.

Here's a Summary

End-user devices with a Mixpanel tracker send event data to Mixpanel's API and these events are pushed onto queues.

This data gets pushed onto Mixpanel's storage system, where storage nodes will write the events to disk using a row-oriented format.

Then, the Compacter service will convert the data from row format to columnar format, making it faster to query.



Given the nature of the work, the Compacter service is very computationally expensive. It runs in an autoscaling nodepool on Google Kubernetes Engine.

When a storage node has a row file of a certain size/age, it will send a request to a randomly selected compacter node to convert it. The compacter node will then return a handle to the resulting columnar file.

If a compacter node has too many requests, then it'll load shed and return an error. The storage node will retry after a backoff period.

A Skew Problem

Mixpanel engineers were having a great deal of trouble scaling the compacter in time to absorb the spikes in load. The compacter service failed to autoscale and this resulted in a spike in errors (as storage node requests were getting shedded and the retries were also getting shedded).

Engineers would have to manually set the autoscaler's minimum number of nodes to a higher number to deal with the load. This resulted in a waste of engineer time and also inefficient provisioning.

When Mixpanel looked at the average utilization of nodes in the compacter service, they expected it to be at 80-90%. This would mean that the compute provisioned in the service was being used efficiently.

However, they found that average CPU utilization was ~40%. They checked the median utilization and the 90th percentile utilization to find that while median utilization was low, the 90th percentile utilization was near 80%.

This meant that half the compacter nodes provisioned were doing little work, while the top 10% of nodes were maxed out.

This was why the autoscaling was messed up, because the autoscaling algorithm was using the average utilization to make its scaling decisions.

Cause for Skew

Engineers were confused about why there was a skew since the storage nodes were randomly selecting compacter nodes based on a uniform random distribution ([Randomized Static load balancing](#)). Each compacter node was equally likely to be selected for a row-to-column conversion job.

However, because the individual jobs had a very uneven distribution in terms of computational load, this caused a large work skew between the compacter nodes.

Mixpanel has a vast range of customers, from startups with thousands of events per day to large companies with billions of events per day.

This meant that the individual jobs were distributed based on a [power law](#), where the largest jobs were significantly larger than the smallest jobs. Some compacter nodes were getting significantly more time-consuming jobs than other nodes and this is what caused the work skew between the nodes.

Having unequal load will also present problems for many other load balancing algorithms as well, like Round Robin.

The Power of 2-Choices

Mixpanel considered several solutions to solve this including inserting a queue between the storage nodes and compacters or inserting a more complex load balancing service. You can check out the [full post](#) to read about these options.

They went with a far simpler solution. They used a popular strategy called [The Power of 2-Choices](#), which uses randomized load balancing.

Instead of the storage nodes randomly picking 1 compacter, they randomly pick 2 compacter nodes. Then, they ask each node for its current load and send the request to the less loaded of the two.

There's been quite a few [papers](#) on this strategy, and it's been found to drastically reduce the maximum load over having just one choice. It's used quite frequently with load balancers like [Nginx](#). Mixpanel wrote some quick [Python simulations](#) to confirm their intuition about how The Power of 2-Choices worked.

Implementing this into their system was extremely easy and it ended up massively closing the gap between the median utilization and the 90th percentile utilization.

Average utilization increased to 90% and the error rate dropped to nearly 0 in steady state since the compacters rarely had to shed load.

For more details, you can read the full summary [here](#).

How Pinterest Load Tests Their Database

Pinterest is a social media service with over 430 million monthly active users. The company relies on advertising for revenue, where they show users promoted posts that are served based on a [real time ad auction](#).

In order to store and serve all of their reporting metrics, Pinterest relies on [Apache Druid](#) - an open source, [column-oriented](#), distributed data store that's written in Java.

Druid is commonly used for [OLAP](#) (analytics workloads) and it's designed to ingest massive amounts of event data and then provide low latency, analytics queries on the ingested data. Druid is also used at Netflix, Twitter, Walmart, Airbnb and many other companies.

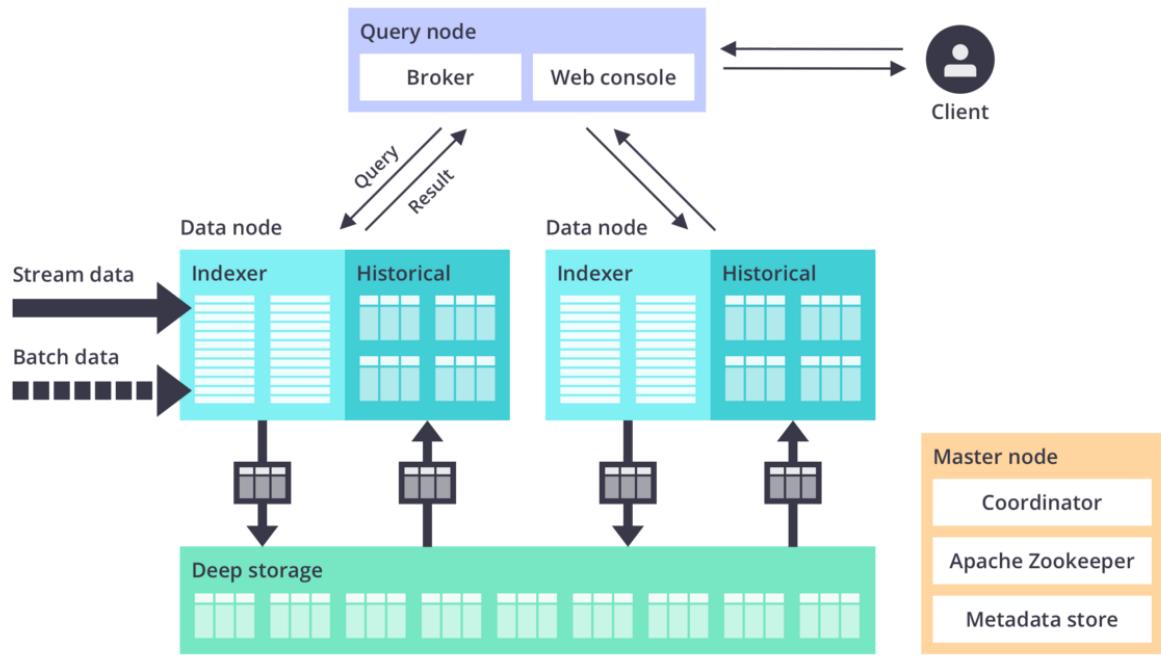
To get an idea of how Druid works, you can split its architecture into three components: the query nodes, data nodes and deep storage.

Deep storage is where the company stores all their data permanently, like AWS S3 or HDFS.

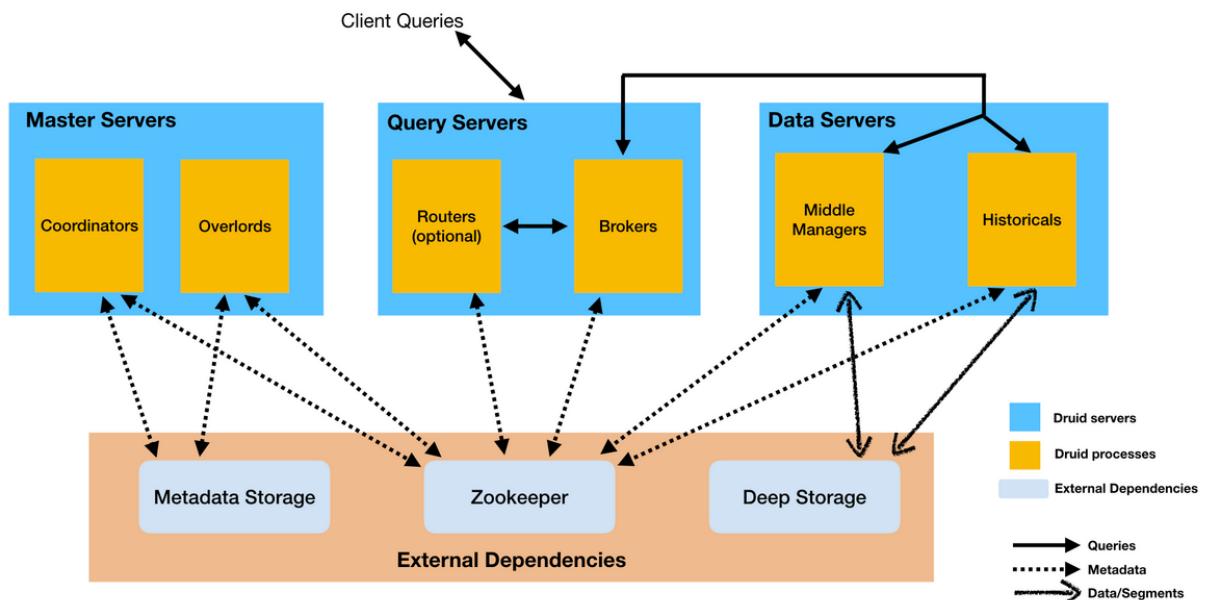
Druid connects with the company's deep storage and indexes the company's data into Druid data nodes for fast analytical queries. New data can also be ingested through the data nodes and Druid will then write it to deep storage.

In order to make analytical queries performant, the data stored in Druid's data nodes is stored in a columnar format. You can read more about this [here](#).

Clients can send their queries (in SQL or JSON) for Druid through the query nodes.



These components are broken down into different services that can be configured and scaled independently.



During the holiday months, Pinterest typically gets a big spike in traffic. In order to deal with this, teams at the company perform extensive load testing in the months prior.

Engineers on the Real-Time Analytics team at Pinterest wrote a great [blog post](#) on the process they go through for load testing Druid.

Here's a summary

When load testing Druid, engineers are looking to verify several areas

1. Queries - The service should be able to handle the expected increase in queries per second and do so within the latency requirements specified in the [service level agreement \(SLA\)](#).
2. Ingestion - The real-time ingestion capabilities should be able to handle the increase in data. Druid should be able to take in all the data and write it to the data nodes and deep storage with low ingestion lag and a low number of failed writes.
3. Data Size - The storage system should have sufficient capacity to handle the increased data volume.

We'll go through each of these and talk about how Pinterest tests them.

Setting up the Testing Environment

When load testing their Druid system, Pinterest can either do so with generated queries or with real production queries.

With generated queries, queries are created based on the current data set in Druid. This is fairly simple to run and does not require any preparation. However, it may not accurately show how the system will behave in production scenarios since the generated queries might not be representative of a real world workload (in terms of which data is accessed, query types, edge cases).

Another option is to capture real production queries and re-run these queries during testing. This is more involved as queries need to be captured and then updated for the

changes in the dataset/timeframe. However, this is more reflective of what Druid will experience.

Pinterest moved ahead with using real production queries and implemented query capture using Druid's logging feature that automatically logs any query that is being sent to a **Druid broker host** (you send your query to a Query Server which contains a broker host).

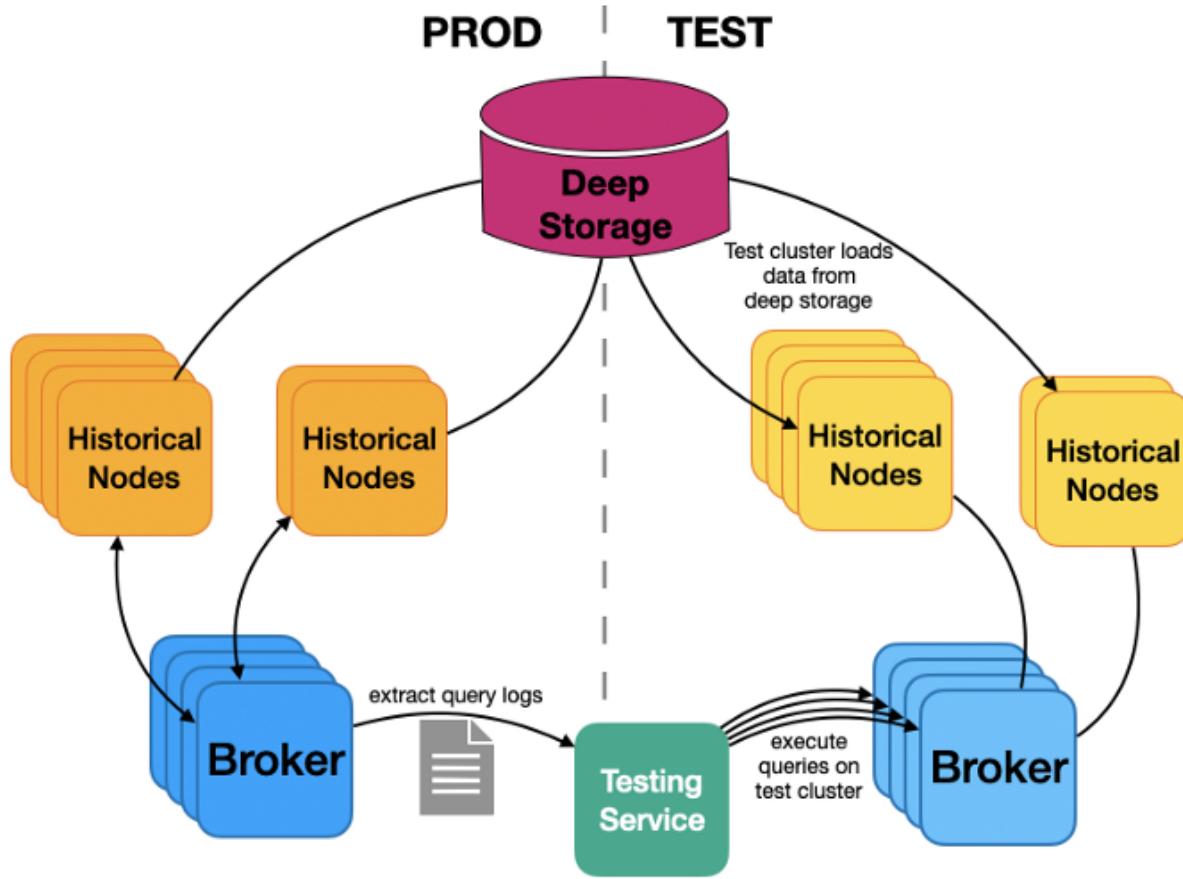
Engineers don't conduct testing on the production environment, as that could adversely affect users. Instead, they create a test environment that's as close to production as possible.

They replicate the Druid setup of **brokers**, **coordinators**, and more and also make sure to use the same host machine types, configurations, pool size, etc.

Druid relies on an external database for **metadata storage** (data on configuration, audit, usage information, etc.) and it supports Derby, MySQL and Postgres. Pinterest uses MySQL.

Therefore, they use a MySQL dump to create a copy of all the metadata stored in the production environment and add that to a MySQL instance in the test environment.

They spin up data nodes in the test environment that read from deep storage and index data from the past few weeks/months.



The testing service loads historical production queries from the log files in the production environment and sends them to the brokers in the test environment for execution. They ramp up the queries per second to test what they expect for holiday traffic.

Evaluating Query Load

Pinterest runs the real production queries on the test environment and looks at several metrics like

- 99th percentile latency of the queries
- CPU usage of the brokers
- Peak queries per second

Using these tests, they can find and address bottlenecks in Druid around the query and data services and adjust how much compute is dedicated for these components.

Some changes can be done quickly while others can take hours. Increasing the number of machines in the query services can be done quickly, whereas increasing the number of data replicas takes time since data needs to be indexed and loaded from deep storage.

Handling Increase in Data Ingestion

Testing Data Ingestion is quite similar to testing queries per second. Pinterest sets up a test environment with the same capacity, configuration, etc. as the production environment.

The main difference is that the Real-Time Analytics team now needs some help from client teams who generate the ingested data to also send additional events that mimic production traffic.

When reviewing ingestion traffic, the Pinterest team looks at

- Ingestion lag
- Number of successful/rejected events
- General system health

And more.

They also make sure to validate the ingested data and make sure it's being written correctly.

Handling Increase in Data Volume

Evaluating if the system can handle the increase in data volume is the simplest and quickest check.

For this, they look at the Druid [Web Console](#), where they can see all the data nodes and current capacity. They estimate the amount of additional data that will be stored over the holiday period and adjust for that.

Results

From the testing, Pinterest found that they were able to handle the additional traffic expected during the holiday period. They saw that the broker pool may need additional hosts if traffic meets a certain threshold, so they made a note that the pool size may need to be increased.

For more details, you can read the full blog post [here](#).

How Facebook Transfers Exabytes of Data Across Their Data Centers Globally

In order to serve their 3 billion users, Facebook runs one of the largest private clouds in the world. They have massive data centers spread out across the globe and they've invested tens of billions of dollars in their infrastructure.

One big challenge when operating at this scale is distributing data across the system. Objects like executable files, search indexes, AI models and containers are a few examples of files that Facebook needs to send to many different machines globally.

Each of these files ranges from a couple of megabytes to a few terabytes and they're split into small chunks. These chunks need to be transferred between Facebook machines with low latency and a very high throughput (millions of machines may need to quickly read a certain object).

These files are stored on Facebook's distributed data store and client machines can read the files from there. However, having all the client machines read from this data store quickly leads to scalability issues.

There are far too many machines requesting files and the transfer speeds would be too slow. Instead, there needs to be a caching system built on top of the distributed data store that can facilitate easy transfer of this data.

To build this system, Facebook tried multiple approaches with varying degrees of centralization. They first tried a highly centralized system with a hierarchical caching layer but that led to scalability issues. They also tried a decentralized approach with the BitTorrent protocol but that was too complex to manage.

Eventually, they settled on a balance between these two approaches with Owl, a system for high-fanout distribution of data objects across Meta's private cloud. Owl distributes over 700 petabytes of data per day to over 10 million unique client machines across Facebook's data centers.

Despite serving over 700 petabytes of data per day, only ~40 petabytes of data is read on average from the underlying data store. This means that Owl has a ~95% cache hit rate and is able to massively reduce the amount of read traffic sent to the underlying data store.

Engineers at Facebook published a great [paper](#) where they talked about their prior data distribution systems (hierarchical caching, bittorrent and more), lessons learned and the architecture/implementation details behind Owl.

Here's a Summary

Facebook engineers needed a way to distribute large objects across their private cloud. The task can be described by 3 dimensions

- Scale - The same object could be read by anywhere from a handful of client machines to millions of clients around the world.
- Size - Objects range in size from 1 megabyte to a few terabytes. Objects are split up into chunks and stored in a distributed storage system.
- Hotness - All the client machines may request the object within a few seconds of each other, or their reads could be spread out over a few hours

Distributing these files must also be done efficiently and reliably. To be considered reliable, the caching system must successfully complete a large percentage of download requests within a certain latency. It should also not be too burdensome for engineers to maintain the system.

Facebook tried several approaches with varying amounts of centralization in the [control plane](#) and the [data plane](#). The data plane are the machines where the cached data is stored while the machines in the control plane determine which files the data plane nodes should cache/delete and how requests should be routed to nodes in the data plane.

Here are a couple of Facebook's initial approaches.

Hierarchical caching

The first attempt was to add a hierarchical cache system in front of their distributed data stores. This is a pretty standard solution and also relatively simple to implement.

Facebook set aside a dedicated pool of machines to use as the caching layer.

When a client machine needs a certain file, their first request goes to a first-level cache. If there's a cache-miss, then the first-level will request the data from the next level caches in the hierarchy (second-level, third-level, and so on). The final layer is the distributed data store itself.

The data would be stored/evicted in these hierarchies so that the first-level cache held the most requested (hottest) data.

The issue with this approach is that it was too difficult for Facebook to handle load spikes for particular pieces of content. Machines in the caching system would get overloaded and start to throttle requests from the clients and Facebook had trouble provisioning capacity appropriately.

They would either provision for the steady state and miss load spikes or they would provision for load spikes and waste compute/servers.

The centralization of the **data plane** on the dedicated pool of machines was making the system too slow to scale.

Bittorrent

To address these scaling issues, Facebook built a second solution based on the **bittorrent protocol**, a very popular protocol for peer-to-peer file sharing.

With this system, any client that wants to download data becomes a peer in the system (so there were millions of peers). Clients would dedicate whatever resources they had available to sharing their downloaded files to other peers in the network. Trackers maintained a list of all the peers and which file chunks were stored on which peers.

When a new peer wants a certain data chunk, it can get a list of other peers that are sharing that chunk from the trackers. After downloading the chunk, that new peer can start sharing that chunk as well.

This scaled much better than hierarchical caching due to the peer-to-peer nature of the system. When there was a load spike for a particular piece of content, the number of peers sharing that content would automatically increase at a similar rate as the demand.

However, each peer in this system was making its own individual decision on which data to request and share. A machine would only become a peer for a certain file if the machine needed to download that file for its own purposes.

This decentralization of the **control plane** led to an inefficient allocation of resources where cold/stale data weren't getting evicted from the system and hot data wasn't being replicated at the optimal level.

The decentralization also made it very hard to operate and debug. Engineers could not get a clear picture of health and status without aggregating data from a large number of peers.

Owl

Facebook then designed a system that combined the best of both of these approaches with Owl.

Owl has a decentralized data plane and a centralized control plane. Data is stored in a decentralized manner (similar to bittorrent) on all the client machines that are downloading from the system. However, decisions around which client stores what chunk and how data is cached on the various peers are managed more centrally.

To accomplish this, Owl has 3 components

- Peer - A peer is a library that's linked with a client machine that wants to download data from Owl. As the machine is downloading the file, it can share data with other client machines (also peers) that request the file. When a peer

wants to download a file, it asks the Owl Tracker (explained below) where to get the data from.

- Superpeer - Superpeers are dedicated machines that can cache and serve data but aren't linked to a client process. Instead, the entire machine is dedicated to caching and sharing data to other peers/superpeers. Owl Trackers will manage what data gets stored on the superpeers.
- Tracker - The trackers are the brain of the system. They tell the peers and superpeers what data to cache/evict based on the entire state of the system. Trackers have a global view of what data is being requested so they can intelligently manage the system.

When a client machine wants to download a file from Owl, it will use the Owl library to send a remote procedure call for the data to a tracker.

The tracker has a global view of state and it will return information on the optimal peer/superpeer that has the data and can share it with the client. The client can then download the data from that node and become a peer itself.

The selection policy for how the tracker selects the peer/superpeer to share the data depends on a variety of factors like geographic distance, load, amount of the file that the node has saved.

If none of the peers have the file, then the tracker can have a superpeer fetch the data from the underlying data store. The superpeer can then share the data with the client machine, making the client machine a peer that can share the file.

The default cache eviction policy is **LRU** where the least recently used files get evicted when a peer's storage is full. However many nodes also use a least rare policy where files are evicted from a peer based on how many other peers have that file cached. A file that is cached on many other peers will be evicted over a file cached on only a few peers.

The system can also be configured to use a hybrid policy of least-rare eviction for hot data and LRU eviction for cold data.

Owl has over 10 million peers and approximately 800 super peers. These are managed by 112 tracker nodes.

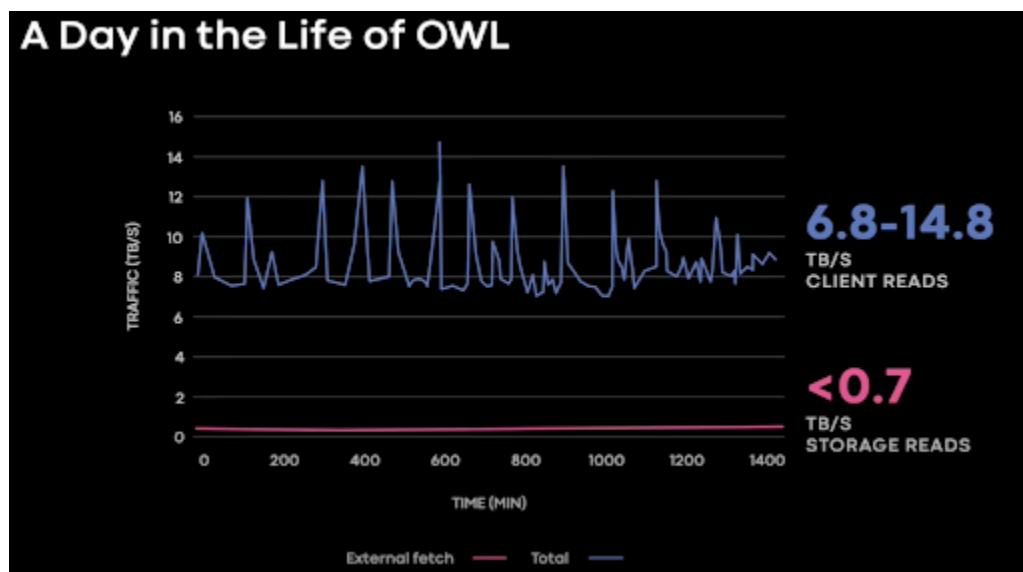
This is just a brief overview of how Owl works. You can get way more details on sharding, security, fault tolerance and much more by reading the full [paper](#).

Results

Facebook started Owl 2 years ago and since then they've seen 200x growth in the amount of traffic Owl is getting. This growth came from replacing the prior systems and taking on their load as well as organic adoption.

Despite this massive increase in traffic, the number of machines needed to run Owl (for the superpeers and trackers) only increased by 4x. The decentralized nature of the data plane (with peer-to-peer distribution) makes the system much easier to scale.

Owl is now handling over 700 petabytes of data per day and has over 10 million client processes using the system. This amounts to a throughput of ~7 - 15 terabytes per second of data that client processes are reading. With Owl, the amount of storage reads that have to be served by the underlying distributed data store is less than 0.7 terabytes per second.



During a typical day, Owl clients will read 700+ petabytes of data per day, but only ~40 petabytes will be read from the underlying data store, equating to a ~95% cache hit rate.

For more details, you can read the full paper [here](#).

How Dropbox maintains 3 Nines of Availability

Dropbox is a file hosting company that stores exabytes of data for their 700 million users. The majority of their users are consumers but many enterprises also use Dropbox's storage solutions.

It's extremely important that Dropbox meets various service level objectives (SLOs) around availability, durability, security and more. Failing to meet these objectives means unhappy users (increased churn, bad PR, fewer sign ups) and lost revenue from enterprises who have service level agreements (SLAs) in their contracts with Dropbox.

The availability SLA in contracts is 99.9% uptime but Dropbox sets a higher internal bar of 99.95% uptime. This translates to less than 21 minutes of downtime allowed per month.

In order to meet their objectives, Dropbox has a rigorous process they execute whenever an incident comes up. They've also developed a large amount of tooling around this process to ensure that incidents are resolved as soon as possible.

Joey Beyda is a Senior Engineering Manager at Dropbox and Ross Delinger is a Site Reliability Engineer. They wrote a great [blog post](#) on incident management at Dropbox and how the company ensures great service for their users.

Here's a Summary

With any incident, there's 3 stages that engineers have to go through.

1. Detection - identify an issue and alert a responder
2. Diagnosis - the time it takes for responders to root-cause an issue and identify a resolution approach.
3. Recovery - the time it takes to mitigate the issue for users once a resolution approach is found.

Dropbox went into each of these stages and described the process and tooling they've built.

Detection

Whenever there's an incident around availability, durability, security, etc. it's important that Dropbox engineers are notified as soon as possible.

To accomplish this, Dropbox built Vortex, their server-side metrics and alerting system.

You can read a detailed blog post about the architecture of Vortex [here](#). It provides an ingestion latency on the order of seconds and has a 10 second sampling rate. This allows engineers to be notified of any potential incident within tens of seconds of its beginning.

However, in order to be useful, Vortex needs well-defined metrics to alert on. These metrics are often use-case specific, so individual teams at Dropbox will need to configure them themselves.

To reduce the burden on service owners, Vortex provides a rich set of service, runtime and host metrics that come baked in for teams.

Noisy alerts can be a big challenge, as they can cause [alarm fatigue](#) which will increase the response time.

To address this, Dropbox built an alert dependency system into Vortex where service owners can tie their alerts to other alerts and also silence a page if the problem is in some common dependency. This helps on-call engineers avoid getting paged for issues that are not actionable by them.

Diagnosis

In the diagnosis stage, engineers are trying to root-cause the issue and identify possible resolution approaches.

To make this easier, Dropbox has built a ton of tooling to speed up common workflows and processes.

The on-call engineer will usually have to pull in additional responders to help diagnose the issue, so Dropbox added buttons in their internal service directory to immediately page the on-call engineer for a certain service/team.

They've also built dashboards with Grafana that list data points that are valuable to incident responders like

- Client and server-side error rates
- RPC latency
- Exception trends
- Queries Per Second

And more. Service owners can then build more nuanced dashboards that list team-specific metrics that are important for diagnosis.

One of the highest signal tools Dropbox has for diagnosing issues is their exception tracking infrastructure. It allows any service at Dropbox to emit stack traces to a central store and tag them with useful metadata.

Developers can then view the exceptions within their services through a dashboard.

Recovery

Once a resolution approach is found, the recovery process consists of executing that approach and resolving the incident.

To make this as fast as possible, Dropbox asked their engineers the following question - *“Which incident scenarios for your system would take more than 20 minutes to recover from?”*.

They picked the 20 minute mark since their availability targets were no more than 21 minutes of downtime per month.

Asking this question brought up many recovery scenarios, each with a different likelihood of occurring. Dropbox had teams rank the most likely recovery scenarios and then they tried to shorten these scenarios.

Examples of recovery scenarios that had to be shortened were

- *Promoting standby database replicas could take more than 20 minutes* - If Dropbox lost enough primary replicas during a failure, then they might be forced to break their availability target if they had to promote a standby replica to primary. Engineers solved this by improving the tooling that handled database promotions.
- *Experiments and Feature Gates could be hard to roll back* - If there was an experimentation-related issue, this could take longer than 20 minutes to roll back and resolve. To address this, engineers ensured all experiments and feature gates had a clear owner and that they provided rollback capabilities and a playbook to on-call engineers.

For more details, you can read the full blog post [here](#).