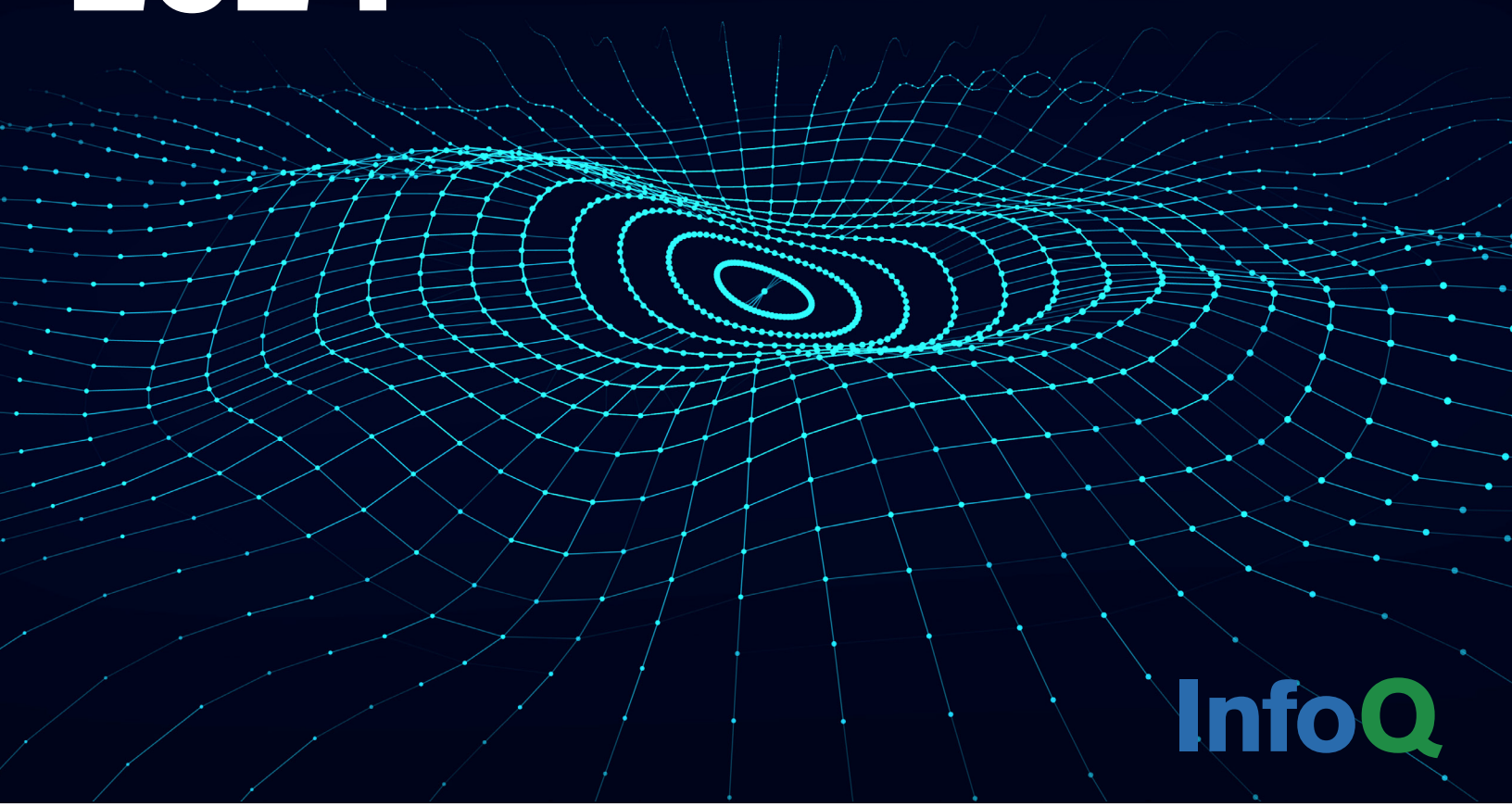
The InfoQ eMag / Issue #111 / April 2024

Architectures You've Always Wondered About 2024



InfoQ

AWS Lambda Under the Hood

Managing 238M Memberships at Netflix

How Ads Ranking Works @Pinterest

InfoQ @ InfoQ InfoQ InfoQ

Architectures You've Always Wondered About 2024

IN THIS ISSUE

PRODUCTION EDITORS Anamaria Bota & Roxana Bacila / COPY EDITORS Maureen Spencer & Susan Conant
GENERAL FEEDBACK feedback@infoq.com / ADVERTISING sales@infoq.com / EDITORIAL editors@infoq.com

CONTRIBUTORS

Mike Danilov

is a Senior Principal Engineer with AWS Lambda. He is currently focused on virtualization technologies, which would enable next generation of execution environments to run customer code on-demand. Mike works on various projects across multiple technologies and time zones. Mike has been with AWS since 2013 and worked in EC2 Virtual Private Cloud organization before joining AWS Lambda.

Surabhi Diwan

builds large scale distributed systems by day and paints by night. She has spent the last decade building and operating complex software systems in insurance, advertising tech at Yahoo, mission critical financial software at Deutsche Bank and doing cloud management at Vmware. She has a Masters in Computer Science from Georgia Tech and currently works as a Sr. Software Engineer at Netflix.

Vignesh Ravichandran

is the Co-founder of Omnigres. Earlier he lead the database team at Cloudflare, Ticketmaster, and Ford. Author of Spinup, pg_savior, and numerous open-source projects.

Anthony Alford

is a Director, Development at Genesys where he is working on several AI and ML projects related to customer experience. He has over 20 years experience in designing and building scalable software. Anthony holds a Ph.D. degree in Electrical Engineering with specialization in Intelligent Robotics Software and has worked on various problems in the areas of human-AI interaction and predictive analytics for SaaS business optimization.

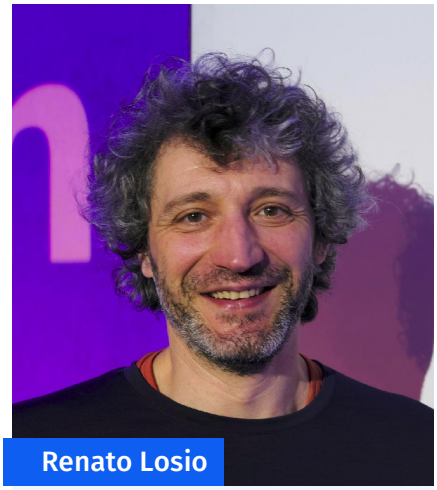
Alex Strachan

is a Staff Software Engineer working on Rippling's regional cell-based architecture and next generation identity framework. Prior to joining Rippling he's tackled a number of problems from scaling Lob's print and mail API to handle the volume from Fortune 100 customers to helping Minted re-model their products to 1000x reduce the client data requirements. His expertise lies in building engineering solutions to unlock business scale; both in the human and technical domains.

Justin Kwan

is a software engineer at Apple. He is currently focused on building high-throughput DDoS mitigation and privacy-preserving features that protect Private Relay and iCloud edge networks. Formerly at Cloudflare, Justin focused on multitenant resource isolation, storage reliability, chaos experimentation across distributed storage infrastructure, and patching the performance of large open-source systems such as Postgres and PgBouncer.

A LETTER FROM THE EDITOR



Renato Losio

Renato has extensive experience extensive experience as a cloud architect, tech lead, and cloud services specialist. Currently, he lives between Berlin and Trieste and works remotely as a principal cloud architect. His primary areas of interest include cloud services and relational databases. He is an editor at InfoQ and a recognized AWS Data Hero. You can connect with him on LinkedIn.

How does Netflix manage its large-scale membership platform with over 238 million subscribers? How can Pinterest scale its social media advertising? While there are still servers involved in serverless architecture, how does AWS operate them to provide the most popular scalable compute service? Furthermore, how does Cloudflare effectively manage a distributed relational database, handling over 55 million row operations per second?

While the scale of these projects may appear extreme, valuable lessons can be gleaned to assist any enterprise seeking to deliver services or engage with business partners, fostering scalable and highly available architectures. The organizations mentioned cover diverse business scenarios, all of which challenge the conventional boundaries of software architectures with metrics and challenges seldom encountered in smaller companies.

“AWS Lambda under the Hood” by Mike Danilov delves into the mechanics of the serverless compute service, which executes code as a highly available, scalable, secure, and fault-tolerant service. The focus lies on Lambda’s routing layer, which contributes to improved system availability and scalability. As AWS Lambda approaches its 10th anniversary this year, Danilov explains how the challenge of cold starts was successfully solved and showcased the introduction of Firecracker, a technology that significantly heightened efficiency while upholding robust security measures.

In “Managing 238M Memberships at Netflix”, Surabhi Diwan discusses the journey of Netflix’s membership team as they navigated the transition from serving a few hundred thousand to 238 million subscribers. The article shows the challenges faced while re-architecting and re-building Netflix apps all while continuing to uphold a seamless member experience, a challenge that Diwan describes as similar to “changing the engine of an airplane while in flight”.

Cloudflare operates a distributed cross-region database architecture, distributing PostgreSQL across multiple regions for

resilience and quick failovers: “Relational Data at the Edge: How Cloudflare Operates Distributed PostgreSQL Clusters” by Vignesh Ravichandran and Justin Kwan highlights how data storage and access at the edge deliver massive performance gains by reducing location-sensitive latency. Ravichandran and Kwan also explore the future trajectory of edge storage, envisioning a transition towards embedding data at the edge and integrating storage with compute resources. Moreover, they address the challenge of data localization, particularly in Europe, signaling ongoing complexities in this domain.

The key to effective data residency is also the focus of “Understanding Architectures for Multi-Region Data Residency” by Alex Strachan: recognizing the strengths and limitations of accelerators and true global databases is crucial for data residency success. Understanding challenges in active-active topologies and conflict resolution methods highlights the need for a nuanced approach beyond relying solely on database proxies.

Finally, “Unpacking how Ads Ranking Works at Pinterest” dives into the dynamic world of social media advertising and describes how Pinterest built its Ad Serving/Ranking infrastructure. The article describes the journey from linear regression to deep learning models, discussing the challenges faced in industrial-scale model creation and how to meet low-latency requests.

We hope that you find value in the articles and resources in this eMag and are inspired by the complex problems faced by some of the most well-known companies in software. We would love to receive your feedback via editors@infoq.com or on Twitter about this eMag. I hope you have a great time reading it!

AWS Lambda under the Hood

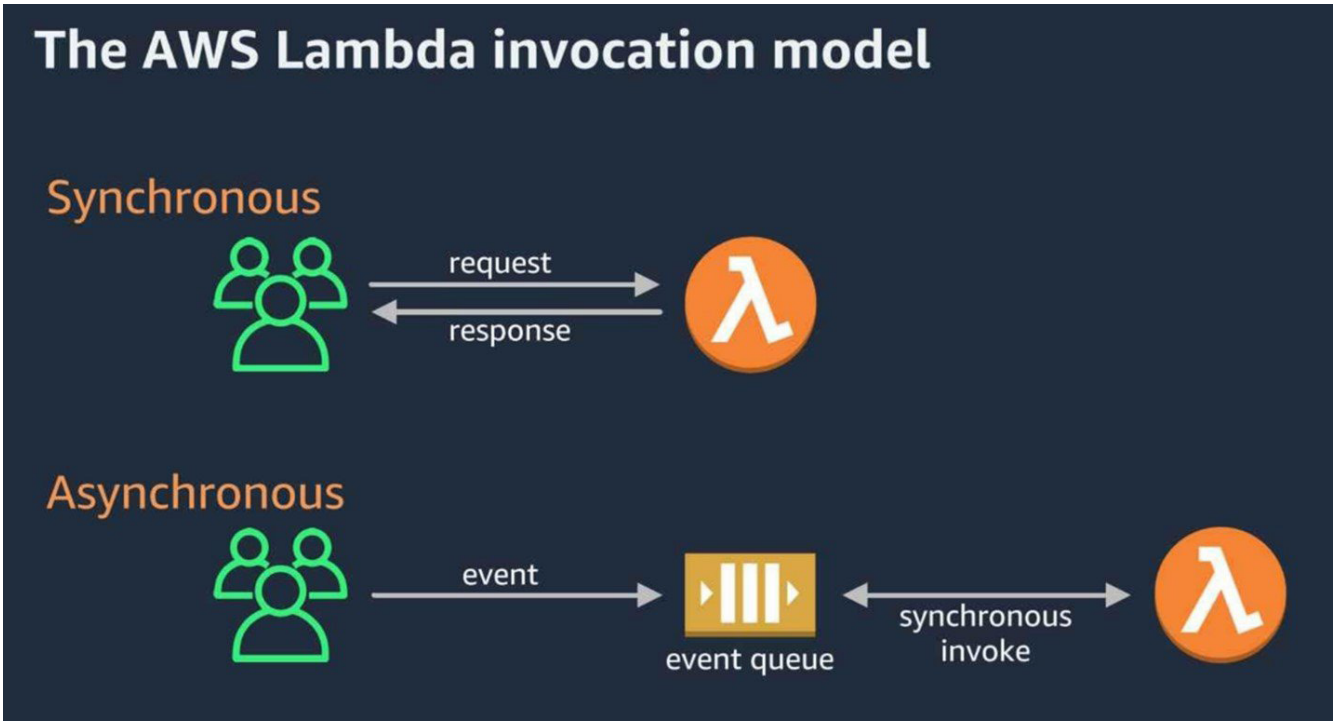
by **Mike Danilov**, Senior Principal Engineer @AWS Lambda

AWS Lambda is a serverless compute service that runs code as a highly available, scalable, secure, fault tolerant service. Lambda abstracts the underlying compute environment and allows development teams to focus primarily on application development, speeding time to market and lowering total cost of ownership.

Mike Danilov, a senior principal engineer at AWS, presented on AWS Lambda and what is under the hood during QCon San Francisco 2023. This article represents the talk, which will start with an introduction to Lambda itself to outline the key concepts of the service and its fundamentals, which will facilitate a deep dive into the understanding of the system.

Subsequently, we will delve into the invoke routing layer, recognized as a crucial component connecting all microservices and ensuring the seamless operation of the entire system. Following that, we will shift towards the compute infrastructure - the space where code execution occurs. This represents a serverless environment within the broader serverless framework. Concurrently, we will weave in a story about cold starts, a common

6



topic when running code in cloud infrastructure.

AWS Lambda Overview

Lambda enables users to execute code on demand as a serverless computing system without needing server ownership, provisioning, or management. Built with various integrated languages, Lambda streamlines the process by allowing users to focus solely on their code, which is executed efficiently.

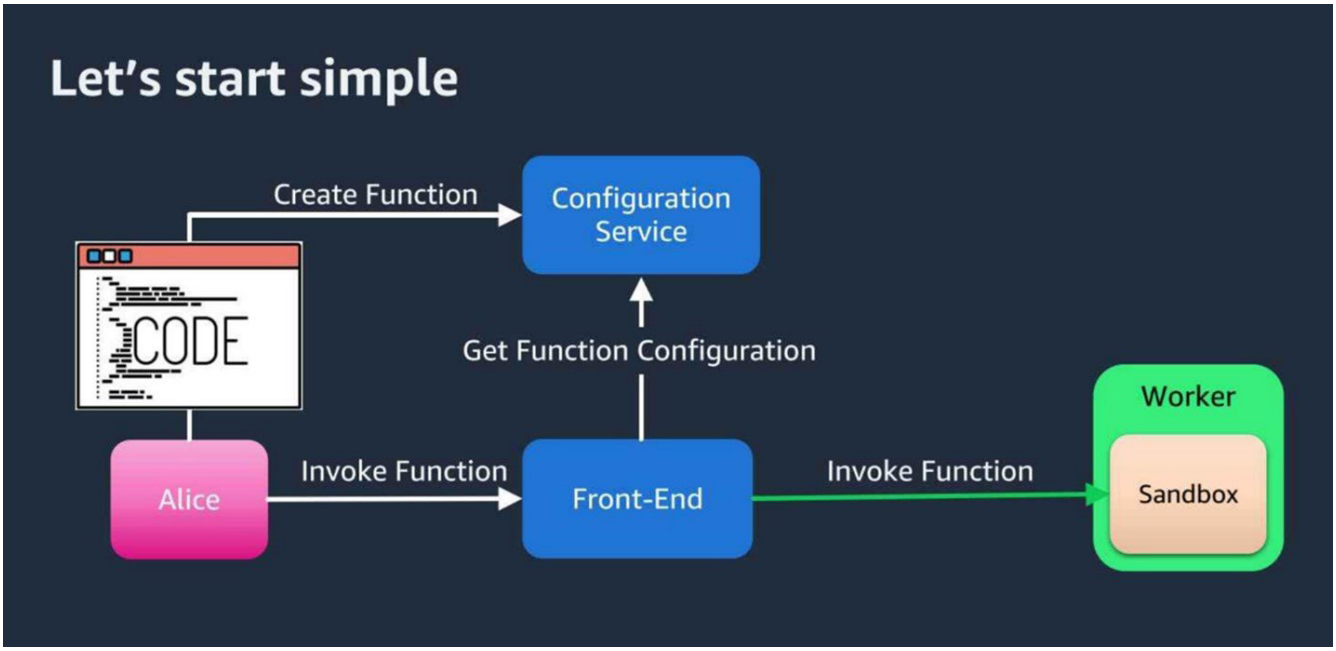
With a rapid response to varying demand, Lambda, which has been in operation for several years, caters to millions of monthly users, generating a substantial volume of trillions of invokes. The operational mechanism involves simple configuration, where users specify their preferred memory size and proportional allocation of resources, including computing and CPU.

Lambda supports two invocation models, starting with the synchronous invoke. In this scenario, a request is sent, routed to the execution environment, and code is executed, providing a synchronous response on the same timeline. On the other hand, asynchronous invoke involves queuing the request, followed by execution at a different timeline through poller systems. Emphasizing the equivalence of execution in synchronous and asynchronous invokes, the discussion primarily focuses on synchronous invokes in the present context.

Its design principles are crucial to understanding Lambda's approach, guiding the framework in making technical decisions and trade-offs. The first tenet, availability, ensures a reliable response to every user request. Efficiency is vital in on-demand systems, requiring quick resource provisioning and release to prevent wastage. Scaling rapidly in response to demand and efficiently scaling down to minimize wastage represents the scale tenet. Security is the top priority at AWS, assuring users a safe and secure execution environment to run and trust their code. Lastly, Lambda emphasizes performance, aiming to provide minimal overhead on top of application business logic, resulting in an invisible and efficient compute system.

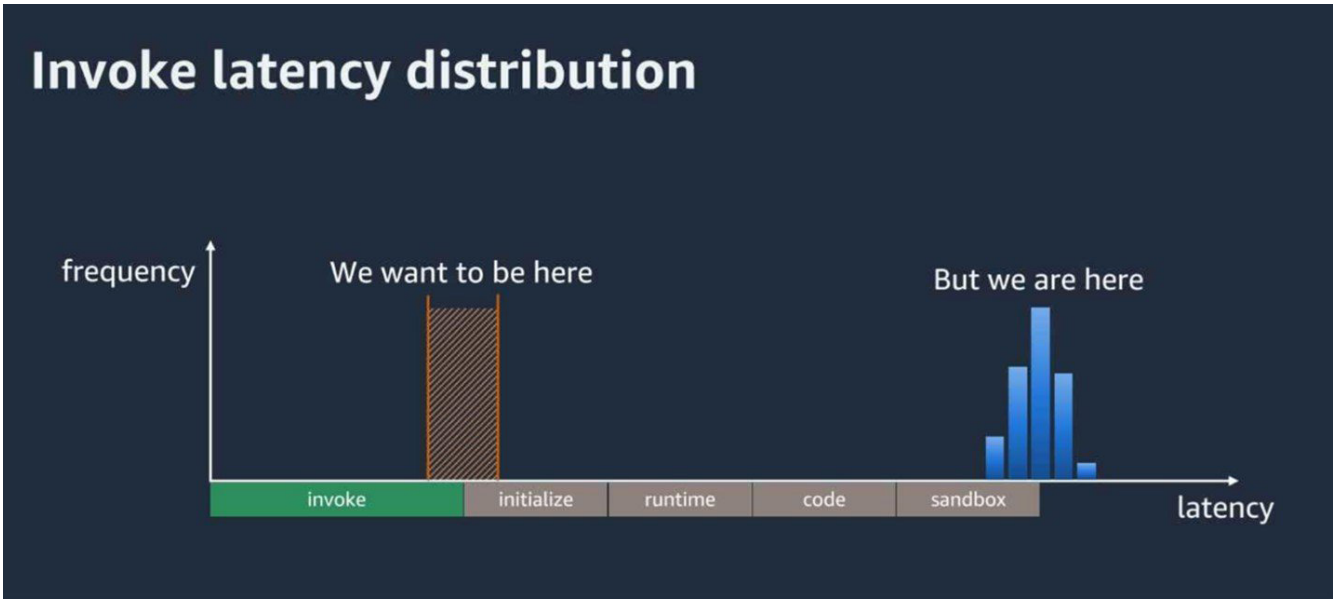
Invoke Request Routing

Invoke Request Routing is a vital part of Lambda - a critical layer interconnecting various microser-



vices, offering essential attributes such as availability, scale, and access to execution environments. Let's take a practical approach by illustrating the building process to understand this layer better.

The scenario involves Alice seeking assistance to deploy her code in the cloud. The initial step is to incorporate a configuration service to store her code and related configurations. Subsequently, a frontend is introduced, which is responsible for handling invoke requests, performing validation and authorization, and storing configuration details in a database. The next component required is a worker, serving as the execution environment or sandbox for Alice's code. Despite the apparent simplicity, challenges arise in an on-demand computing system, where the availability of workers or execution environments may be uncertain during an invoke. To address this, a new system called "placement" is introduced to create execution environments or sandboxes on-demand.

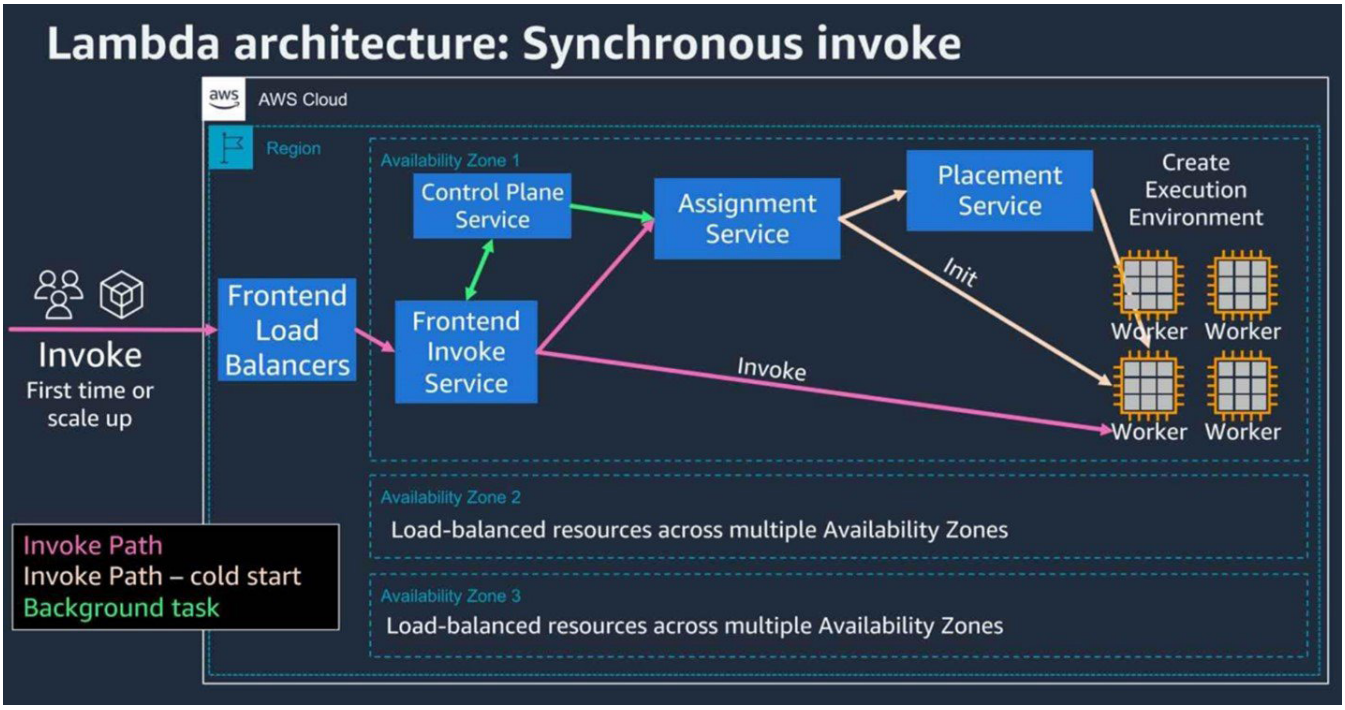


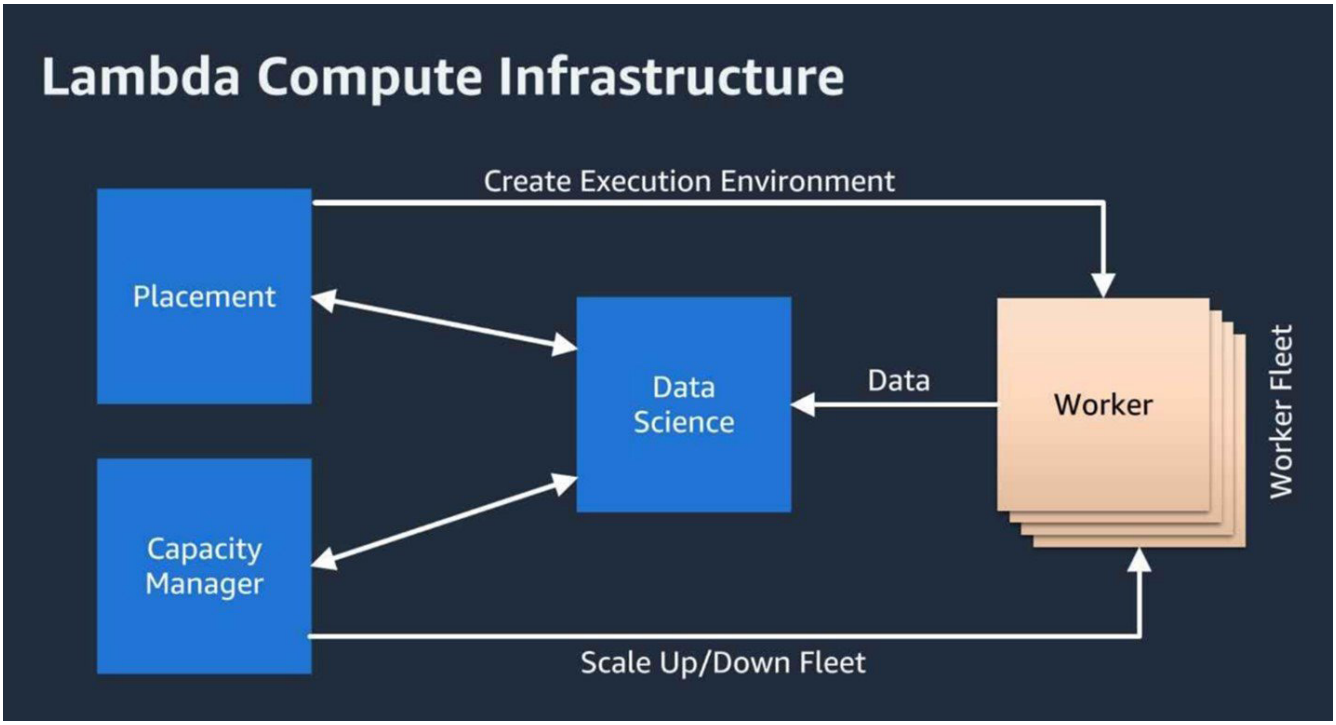
The front end needs to communicate with placement to request a sandbox before forwarding the invoke request to the created execution environment, completing the functional setup. However, a challenge persists due to the inclusion of on-demand initialization before each invoke request. This initialization involves multiple steps, including creating the execution environment, downloading customer code, starting a runtime, and initializing the function. This process can take several seconds, potentially negatively impacting the overall customer experience.

The latency distribution graph illustrates the frequency of invoking durations over time. The green latency represents successful code execution, reflecting the efficiency of the business logic and its CPU utilization. To minimize overhead and enhance customer experience, a new system, the worker manager, is introduced. Operating in two modes, it either provides a pre-existing sandbox upon frontend request, leading to smooth "warm invokes," or, in the absence of a sandbox, initiates a slower path involving placement to create a new one. While warm invokes exhibit minimal overhead and speed, efforts are underway to eliminate cold starts, acknowledging the need for further improvements.

Above presents a practical depiction of Lambda synchronous invoke routing in production, emphasizing enhancements for resilience. Additional availability zones and a front load balancer were integrated. The worker manager, responsible for tracking sandboxes, posed challenges due to its reliance on in-memory storage, leading to potential data loss in case of host failures. A replacement named the assignment service was introduced a year ago to address this. Functionally similar, it features a reliable distributed storage known as the journal log, ensuring regional consistency.

The assignment service utilizes partitions, each with a leader and two followers, leveraging a leader-follower architecture to facilitate failovers. This transformation significantly bolstered system availability, rendering it fault-tolerant to single host failures and availability zone events. The move from in-memory to distributed storage, coupled with implementing a leader-follower model, improved efficiency and reduced latency. This marks the conclusion of the first chapter on invoke routing, exploring aspects of cold and warm invokes and the





pivotal role of a consistent state in enhancing availability.

Compute Fabric

Compute fabric, specifically the worker fleet within Lambda's infrastructure, is responsible for executing code. This fleet comprises EC2 instances, known as workers, where execution environments are created. A capacity manager ensures optimal fleet size adjustments based on demand and monitors worker health, promptly replacing unhealthy instances. Data science collaboration aids placement and capacity managers in informed decision-making by leveraging real-time signals and predictive models.

Considering data isolation, the text explores the challenge of running multiple users' code on the same worker. Adopting Firecracker, a fast virtualization technology, helps overcome this hurdle. By encapsulating each execution environment in a microVM, Firecracker ensures strong data isolation, allowing diverse accounts to coexist on the same worker securely. This transition from EC2 to Firecracker significantly enhances resource utilization, preventing overloads and providing consistent performance.

The benefits of Firecracker, including robust isolation, minimal system overhead, and improved control over the worker fleet's heat, are highlighted. The adoption of Firecracker results in a notable reduction in the cost of creating new execution environments, as demonstrated in the latency distribution diagram. The narrative then introduces the idea of using VM snapshots to expedite the initialization process, leading to reduced overhead for creating new execution environments.

The process of building a system for VM snapshots is outlined. The text emphasizes distributing snapshots between workers, ensuring fast VM resumption, and maintaining strong security measures. Introducing an indirection layer, known as "copy-on-read," addresses potential security threats associated with shared memory. The challenges of restoring uniqueness to identical VMs and ongoing collaboration with various communities, such as Java and Linux, are discussed as part of the continuing efforts to enhance the security and efficiency of the system.

Snapshot Distribution

Snapshot distribution is a critical aspect of EC2 instances, especially considering the significant size of snapshots, which can reach up to 30 gigabytes. Traditional download methods could be time-consuming, taking at least 10 seconds to complete. An analogy is drawn to video streaming to optimize this process, where content is progressively loaded in the background as the initial portion is played. Similarly, snapshots are split into smaller chunks, typically 512 kilobytes, allowing the minimal set of chunks required for VM resumption to be downloaded first. This approach has the dual advantage of amortizing download time and retrieving only the necessary working set during an invoke.

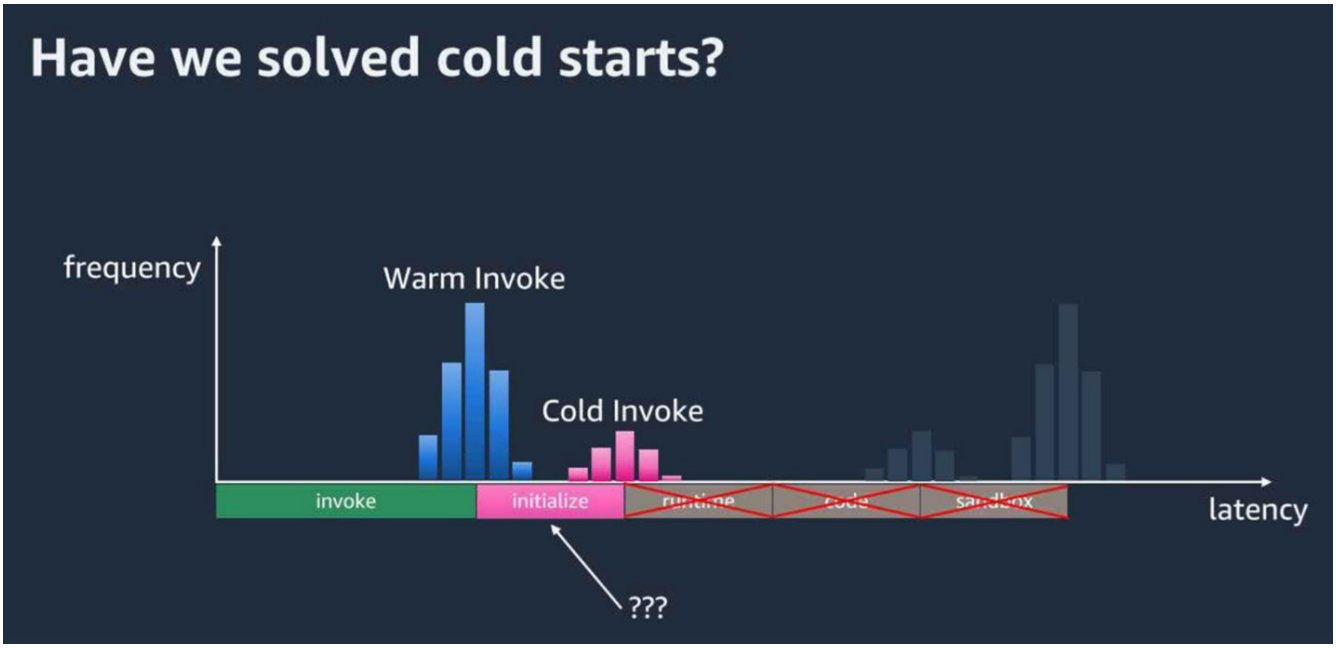
The mechanism for on-demand chunk loading involves mapping VM memory to the snapshot file. When a process accesses memory, it either retrieves data from the memory pages or, if not available, falls back to the snapshot file. The efficiency of this process relies heavily on cache hit ratios, encompassing local, distributed, and origin caches. A strategy is proposed to maximize the cache hit ratio by identifying and sharing familiar chunks. For instance, operating system and runtime chunks can be deduplicated and shared across multiple functions, enhancing efficiency and minimizing calls to the region.

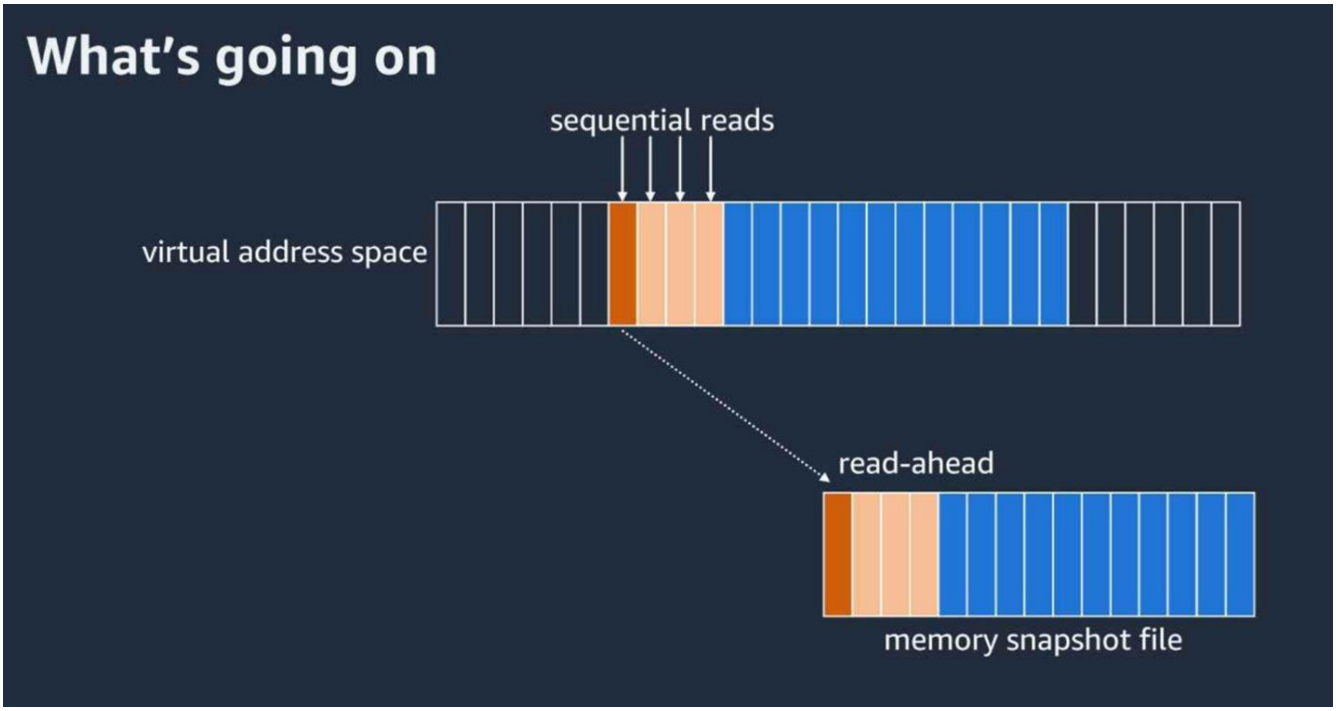
Layered incremental snapshots are suggested to optimize chunk management further. These snapshots are encrypted with different keys, categorized as operating system, runtime, and function chunks. Operating system and runtime chunks can be shared, leveraging convergent encryption to deduplicate common bits even when the origin is unknown. Convergent encryption ensures that identical plaintext content results in equal encrypted chunks. This comprehensive approach enhances cache locality, increases hits to locally distributed caches, and reduces latency overhead.

In the production system, the indirection layer is replaced with a sparse file system, streamlining the request process and providing chunks on-demand at the file system level. This sophisticated approach contributes to a more efficient and responsive system.

Have We Solved Cold Starts?

The system should ideally function well after successfully implementing snapshot distribution and VM resumption. However, some delay persists in sure cold invokes despite their proximity to the target location. To comprehend this issue, revisiting page





caches and memory mapping is essential. An optimization in the operating system involves read-ahead, where, anticipating sequential reads in regular files, multiple pages are read when a single page is accessed. In the case of mapped memory instead of a regular file, which allows random access, this method proves inefficient, leading to the download of the entire snapshot file for seemingly random page requests.

This inefficiency is depicted in the latency distribution graph, highlighting the need for a resolution. To address this, an analysis of memory access among 100 VMs reveals a consistent access pattern. This pattern, recorded in a page access log, is attached to every snapshot. Consequently, during snapshot resumption, the system possesses prior knowledge of the required pages and their sequence, significantly enhancing efficiency. This innovative solution successfully mitigates issues associated with cold starts. Notably, users can experiment with this improvement firsthand by enabling Lambda SnapStart on their Java functions and experiencing the optimized performance of VM snapshots.

Conclusion

In this article, we delved into the invoke routing layer within Lambda, resulting in enhanced system availability and scalability. A comprehensive exploration of the compute infrastructure showcased the introduction of Firecracker, a technology that significantly heightened efficiency while upholding robust security measures. The system's performance was markedly improved, and the challenge of cold starts was successfully solved. These efforts culminated in a fundamental concept: executing code in the cloud without servers, which captures Lambda's essence as a compression algorithm for user experience.

Lastly, more details are available here.

SPONSORED ARTICLE

AI on the Edge: How We Taught Our CDN to Recognize Kittens

Imagine a content delivery network (CDN) that not only swiftly caches and manages data but also interprets it intelligently. That's FastEdge, our groundbreaking edge-computing solution. FastEdge represents a leap in our technological prowess, harnessing our global network more efficiently than ever before. With this new solution, we were able to perform image recognition on our CDN, beginning with the fundamental pillar of the modern internet: kittens. Read on to discover how and why we taught our CDN to recognize kittens with FastEdge.

FastEdge: The Edge of Innovation

FastEdge is a low-latency, high-performance serverless platform designed for the development of responsive and personalized applications leveraging WebAssembly (Wasm) runtime. FastEdge resides directly on our edge servers, and this integration allows you to distribute custom code globally across more than 160 edge locations. The result is a near-instantaneous response to user inputs and exceptional application responsiveness—qualities that are highly valued when building modern distributed apps.

In developing FastEdge, we focused primarily on web optimization and custom network logic applications. However, the growing importance of AI inspired us to explore a new use case. Driven by our own curiosity, we decided to run AI on FastEdge, a significant milestone in the search for innovative computing solutions.

Why Run AI on the Edge?

Imagine you're using an app like ChatGPT. Initially, its AI model gets fed a vast quantity of data: it learns to recognize patterns, understand language, or make predictions based on this data. This phase is called model training; it requires substantial computational resources and is typically done on high-performance servers. A trained AI model can then be deployed on other computer systems.

When you take a trained model and ask it a new question, it quickly gives you an answer. This process is called AI inference. The faster inference happens, the better your experience. And the closer that inference servers are located to end users, the faster inference happens.

With FastEdge operating on our CDN with 160+ points of presence, we realized that FastEdge has many of the infrastructure requirements for AI inference already in place: a flexible and versatile network with global-scale load balancing built upon a modern software stack.

Challenges

As we ventured into integrating AI capabilities at the edge using FastEdge, we revealed a fundamental challenge: while Wasm shines in traditional web tasks due to its speed, compactness, and security, it isn't optimized for the demands of running AI models.

Wasm excels in environments where tasks are lightweight and execution times are short. AI inference, in contrast, often demands extensive computational power, typically relying on GPUs for faster processing. CPUs alone—the hardware on which FastEdge runs—often struggle to meet these demands.

Please read the full-length version of this article here.

Relational Data at the Edge: How Cloudflare Operates Distributed PostgreSQL Clusters

by **Vignesh Ravichandran**, Co-founder and CEO of Omnigres
and **Justin Kwan**, Software Engineer - iCloud Private Relay @Apple

This is a summary of a talk we gave at QCon San Francisco in October 2023 where we discussed the high availability setup and considered the tradeoffs that Cloudflare had to make around each part of the system. We will explore some of the performance challenges that Cloudflare faced when bringing the database infrastructure closer than ever to the edge and dive deep into the solutions that have been implemented.

What is Edge?

The edge, in the context of distributed systems, refers to services located geographically close to clients. This includes tasks like DNS resolution, content delivery, and IP firewalling. Traditionally, centralized data centers hosted services like relational databases, which were farther from clients. Proximity to clients defines the metric, reducing network latency and costs by serving requests nearer to the client's location.

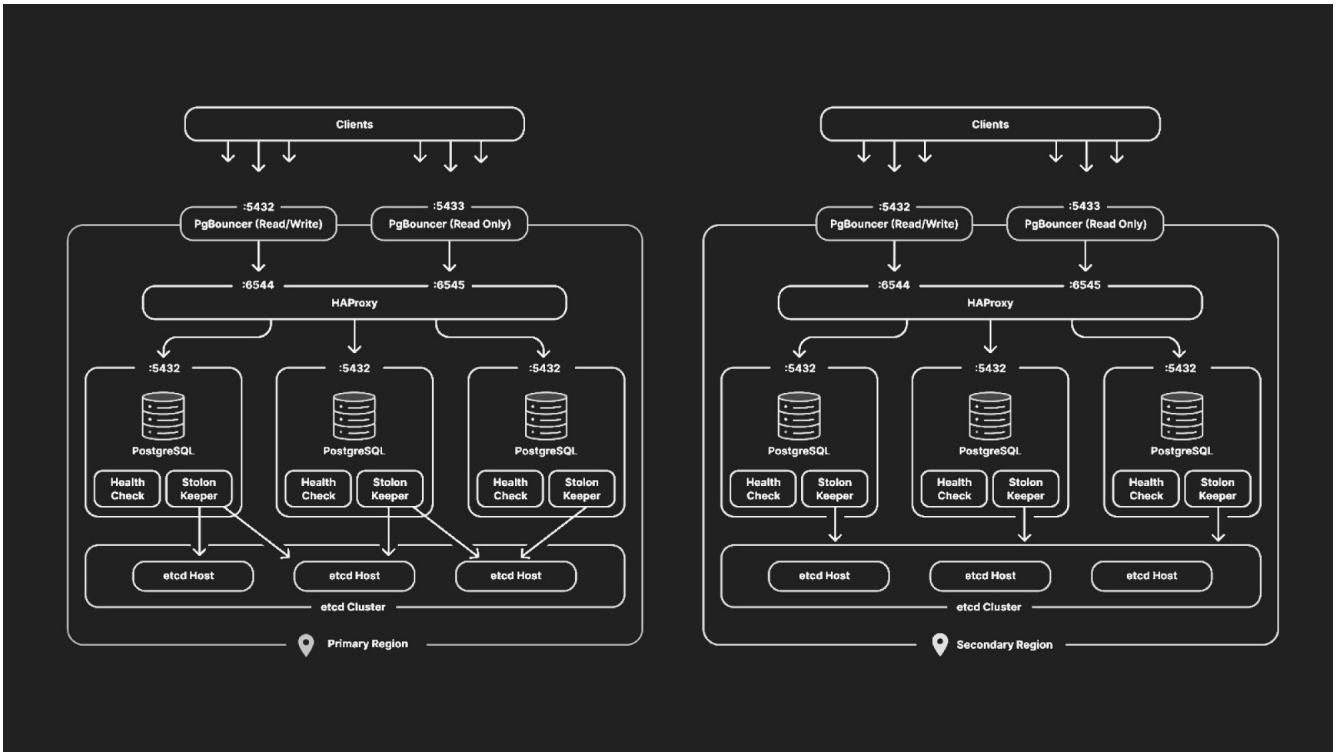


Figure 1: Cloudflare database architecture - image source

The Control Plane

Cloudflare safeguards over 27 million internet properties, managing 46 million HTTP requests per second on average. The network spans 250 global Points of Presence locations, handling over 55 million row operations per second at peak on the busiest PostgreSQL cluster, with over 50 terabytes of data across all clusters.

Cloudflare's control plane orchestrates numerous microservices, powering the dashboard. It manages rules and configurations for crucial network services along the data path. The database clusters store data for each customer such as DNS record changes, firewall rules, DDoS mitigation, API gateway routes, and internal service information like billing entitlements and user authentication. Application teams often leverage PostgreSQL for unique purposes, using stored procedures for executing business logic with transactional consistency. PostgreSQL is also utilized as an outbox queue, capturing domain events such as generated DDoS rules from traffic analysis in a centralized data center.

A separate daemon retrieves these events from the database and channels them through Kafka to edge services, allowing latency-critical services to bypass direct database access and benefiting from the durability provided by PostgreSQL and Kafka.

Cloudflare runs the software and service stack on bare metal, dealing with rack-mounted servers, high-bandwidth network cards, and operational maintenance. On-premise data storage provides maximum flexibility, allowing the team to fine-tune elements like solid-state RAID configuration and enhance the system with an open-source cluster management system. This transparency and fine-grained control of the cluster would be impossible with managed cloud databases like Amazon RDS.

Unfortunately, this implies there is no instant autoscaling button to boost the capacity of the cluster when load increases, an inherent challenge in running the entire stack on-premise.

Architecture

Let's discuss Cloudflare's database architecture at the edge which needs to process transactions on the order of millions per second.

In designing the system, the team prioritizes high availability, aiming for a Service-Level Objective (SLO) of five 9s, with a maximum of five minutes and a half of downtime per year across the entire stack. The transactional workloads lean towards read-heavy operations. The infrastructure has to handle a high rate of read and write operations with minimal latency, as well as maintaining fault tolerance. Cloudflare leverages internally an anycast BGP network, with clients naturally load balanced across the PgBouncer proxy instances.

While BGP Anycast optimizes read queries in proximity to clients, write queries are forwarded to the primary region where the primary database instance resides.

At the top, PgBouncer manages database server connection pools for the application clients. HAProxy then load balances queries across multiple database cluster instances, preventing overload. In a typical setup, a primary instance replicates data to multiple replicas for a high rate of read queries, managed by the stolon high-availability cluster manager backed by etcd. In this setup, etcd is used for distributed cluster leadership consensus and configuration consistency.

The active-standby model ensures cross-region data redundancy, with the primary region in Portland handling inbound queries and the standby region in Luxembourg ready for traffic redirection or failovers.

We will now analyze each layer, exploring components and tradeoffs.

Persistence at the Edge

The decision to build a highly distributed and relational edge database led the team to consider fundamental architecture principles, particularly the CAP theorem. Since the application is geographically distributed and relies on open-source software like PostgreSQL, you have to prioritize either consistency or high availability, you can have only one of the two, usually favoring the latter.

In a single data center, the typical architecture includes a primary database, a synchronous replica, and several asynchronous replicas. This topology is replicated across multiple data centers. The semi-synchronous replication setup ensures that if an asynchronous replica goes down, it does not impact applications significantly. The same tradeoffs apply to cross-region replication, allowing applications in one region to continue working if another region goes down. However, if the synchronous replica fails, the entire application halts to prevent inconsistencies, making it the chosen primary during failovers. This semi-synchronous replication topology balances the requirements effectively.

PostgreSQL, originating from the '90s in Berkeley, was initially designed with a monolithic architecture. To transform it into a distributed system, we heavily rely on two replication aspects: logical replication and streaming replication. Cascading replication, a third option, is built on top of logical and streaming replication.

Before delving into replication's role in creating a distributed cluster, let's briefly touch on write-ahead logs (WAL). In any relational database, including MySQL or Oracle, the durability in ACID is achieved through WAL. Changes to the database, initially saved in volatile memory and asynchronously flushed to disk, are first sequentially recorded to a disk-based WAL file. This ensures data durability via recovery in the case of a crash.

This feature proves vital in building replication systems, conveniently enabling us to capture and replicate units of work (each log entry).

Streaming replication mode is straightforward: a replica establishes a TCP connection and efficiently streams log entries from the primary to another repli-

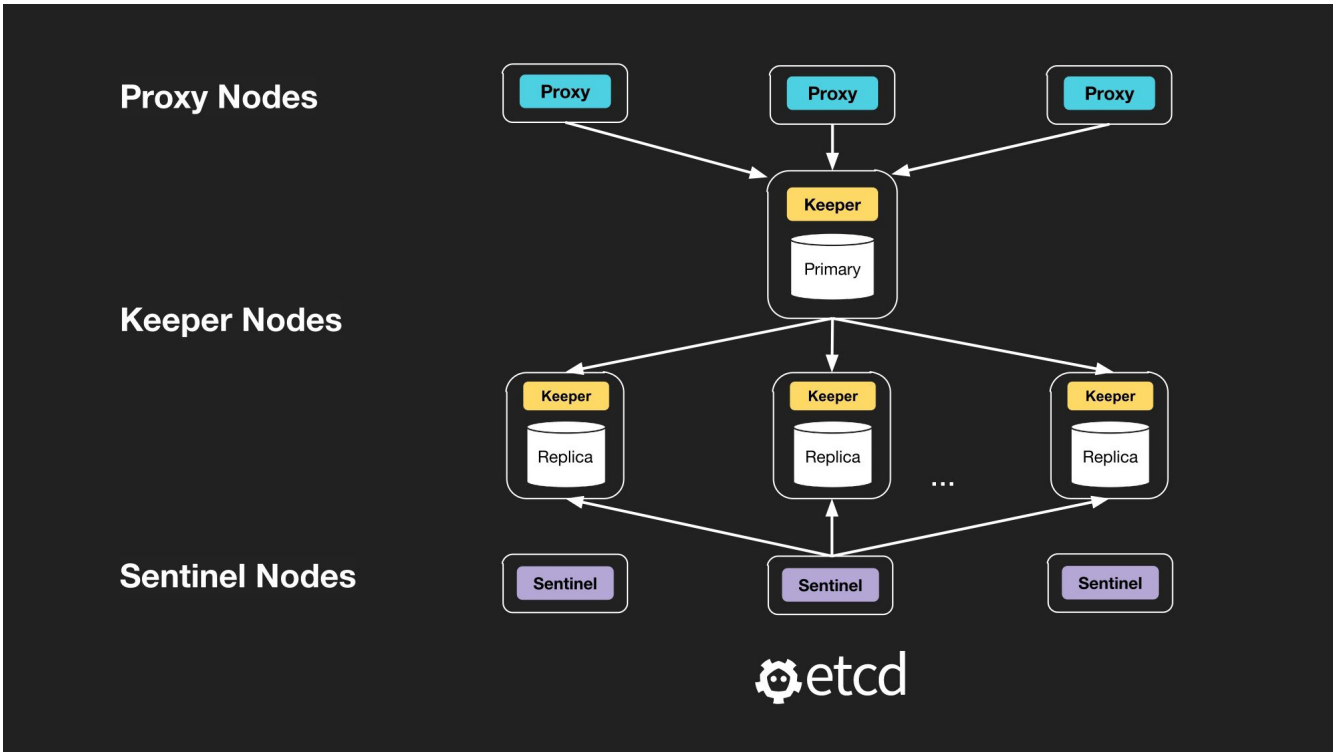


Figure 2: stolon's architecture

ca. The notable advantage is its performance, capturing changes at up to one terabyte per second with minimal delay, though we will cover potential sources of delay later in the article. A caveat is that it›s an all-or-nothing approach, replicating all changes to every Postgres object due to its filesystem-level block replication.

Logical replication, a newer version, operates at the SQL level, providing flexibility in replicating data at the table, row, or column level. However, it is not possible to replicate DDL changes, requiring custom tools, and it has scalability challenges, especially at the multi-terabyte scale.

Cluster Management

One of the primary reasons for cluster management is addressing database failures. Failures, whether logical, like data corruption, or more severe, such as hardware-based failures due to natural disasters, are inevitable. Manual failovers in these situations are cumbersome. Additionally, around 5% of Cloudflare commodity servers are faulty at any point in time, translating to ongoing failures across a fleet of thousands of servers and over multiple data centers. An automated cluster management system is essential for efficiently handling these diverse and critical failures.

The team opted for stolon, an open-source cluster management system written in Go, running as a thin layer on top of the PostgreSQL cluster, with a PostgreSQL-native interface and support for multiple-site redundancy. It is possible to deploy a single stolon cluster distributed across multiple PostgreSQL clusters, whether within one region or spanning multiple regions. Stolon's features include stable failover with minimal false positives, with the Keeper Nodes acting as the parent processes managing PostgreSQL changes. Sentinel Nodes function as orchestrators, monitoring Postgres components› health and making decisions, such as initiating elections for new primaries. Proxy Nodes handle client connections, ensuring a single write primary to avoid multi-master situations.

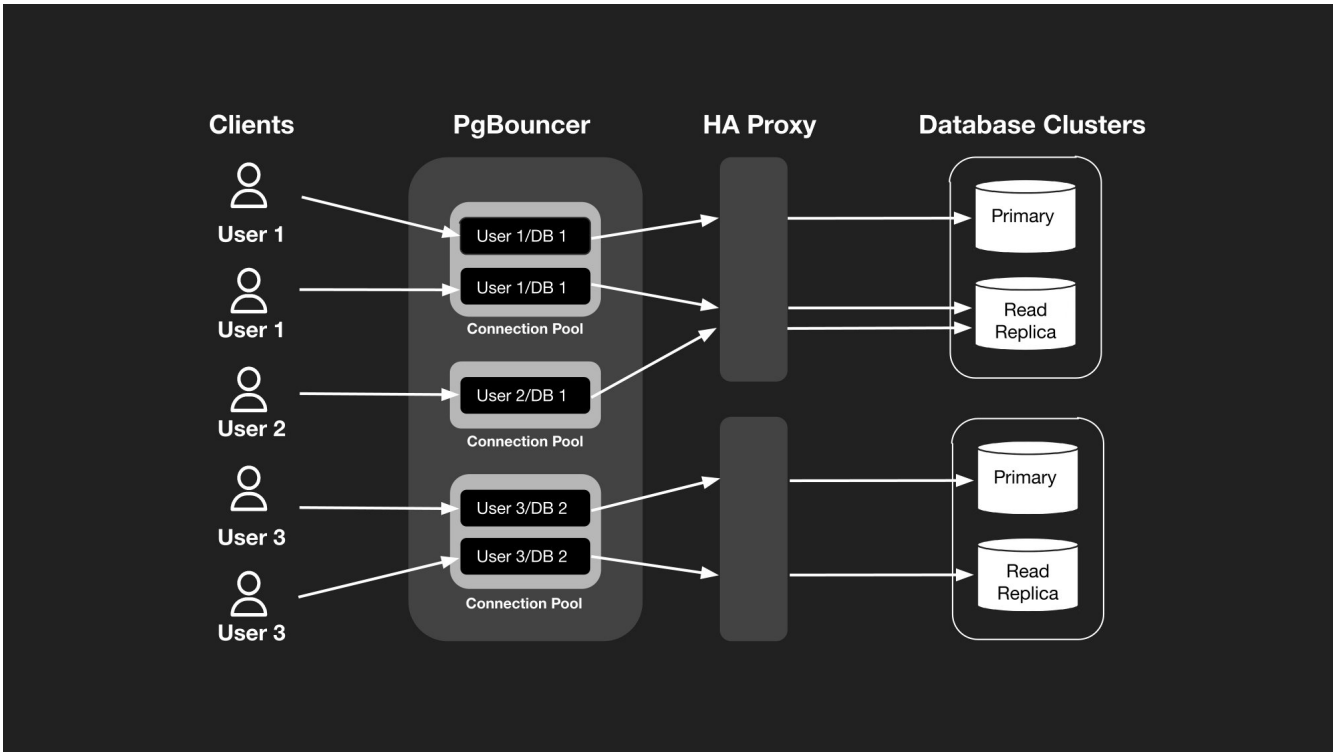


Figure 3: How PgBouncer works

Connection Pooling

Database connections, being finite resources, require efficient management due to the inherent overhead.

PostgreSQL connections rely on the TCP protocol, involving a three-way handshake and an SSL handshake for communication security. Each connection dedicates a separate OS-level process, necessitating forking by the main postmaster process, leading to CPU time consumption and memory usage. As thousands of concurrent connections are opened, socket descriptors and CPU time allocated for transaction processing diminish.

A server-side connection pooler manages a fixed number of database connections while exposing a PostgreSQL-compatible interface to the clients. As an intermediary, it reduces the total open connections by recycling them as clients connect through it to the database.

This centralizes control over tenant resources, allowing the team to regulate the number of allocated connections per upstream application service. Cloudflare's enhancements to PgBouncer include features inspired by TCP Vegas, a TCP congestion avoidance algorithm that enforces stricter multi-tenant resource isolation by restricting tenant total throughput.

Cloudflare chose PgBouncer as the connection pooler due to its compatibility with PostgreSQL protocol: the clients can connect to PgBouncer and submit queries as usual, simplifying the handling of database switches and failovers. PgBouncer, a lightweight single-process server, manages connections asynchronously, allowing it to handle more concurrent connections than PostgreSQL.

PgBouncer introduces client-side connections, separate from direct server PostgreSQL connections, and employs an efficient non-blocking model for network I/O, utilizing a single thread and the OS-provided epoll mechanism for event monitoring.

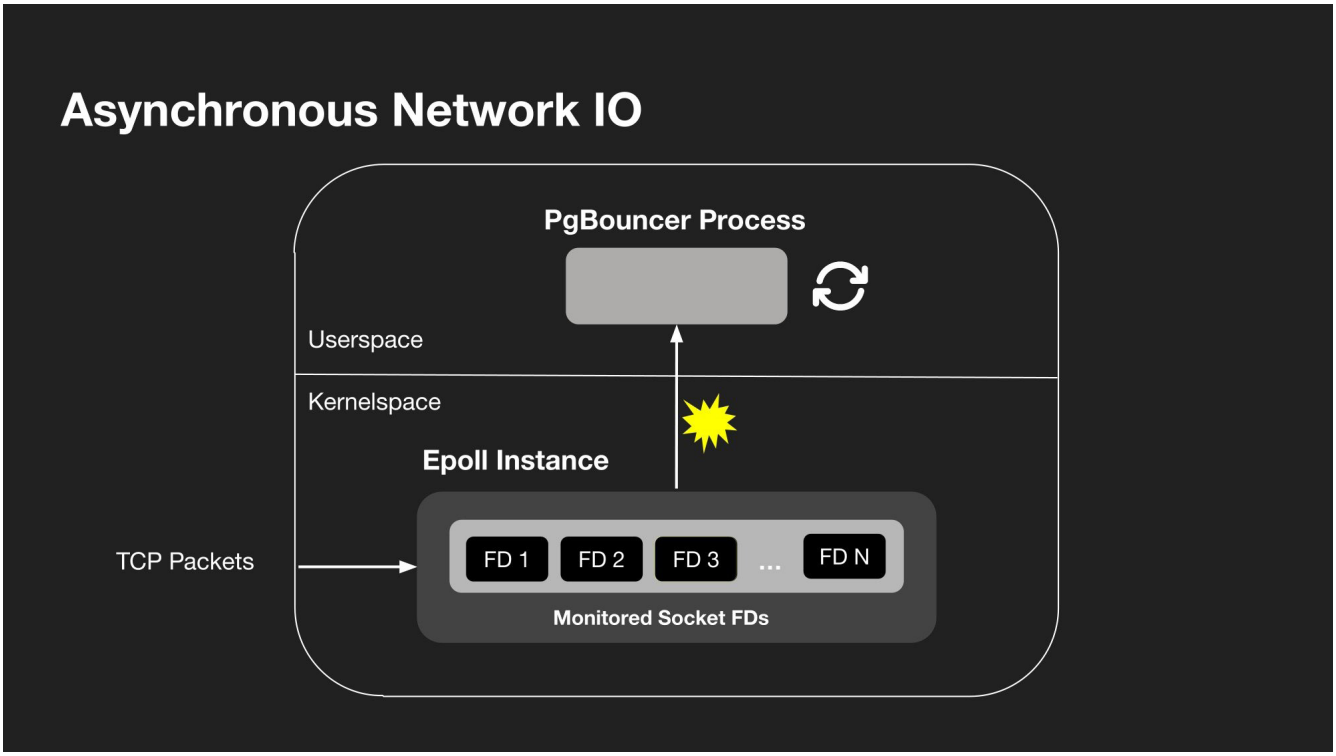


Figure 4: One PgBouncer process

Unlike PostgreSQL's thread-per-connection model, PgBouncer's single-threaded event loop minimizes memory usage by avoiding multiple thread stacks and improves CPU utilization by servicing requests across all clients in one thread, preventing wasted CPU time from idle connection threads.

To address the challenge of enabling multiple single-threaded PgBouncer processes to utilize all CPU cores on a single machine, the solution was found in the operating system: the team added the SO_REUSEPORT socket option to allow multiple process sockets to simultaneously listen on the same port.

Load Balancing

Cloudflare uses the HAProxy load balancer to evenly distribute incoming database queries across multiple PostgreSQL servers, preventing overloads against any single server. Similar to PgBouncer, HAProxy offers high availability and fault tolerance by redirecting traffic from failed servers to healthy ones which reduces downtime from degraded database instances. HAProxy efficiently handles TCP connections at Layer 4 with minimal overhead by using the kernel splice system call. Inbound and outbound TCP streams are attached within the kernel itself, enabling data transfer without the need to copy it into userspace.

Challenges and Solutions

Let's explore challenges in the database infrastructure and share some performance tips for achieving high availability at the edge. Replication lag is especially pronounced under heavy traffic and write-intensive operations like executing ETL jobs. Any bulk write operations, such as data migrations and GDPR compliance deletions, and even automatic storage compaction (autovacuum) also amplify the replication lag.

The team targets a 60-second replication lag SLO, advising application teams accordingly. To minimize the replication lag, SQL query writes are batched into smaller chunks, ensuring

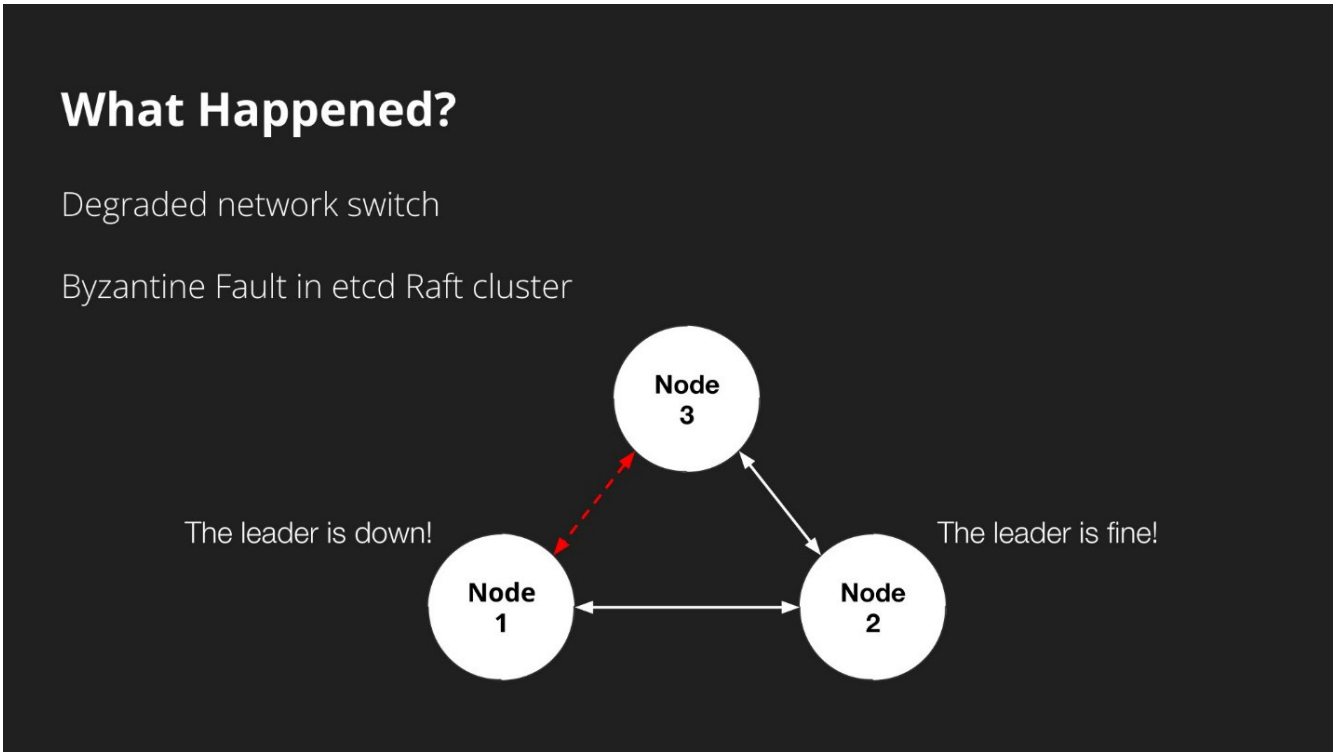


Figure 5: Conflicting information in the cluster

smoother replication. Read-after-write consistency is maintained by caching or reading directly after writing to the primary or to a synchronous replica. An unconventional approach to avoid lag is ejecting all replicas from the cluster, leaving only the primary. While this has been effective in practice, it requires a deep understanding of query workloads and potential system changes. You can think of this as the unsafe keyword in Rust.

In 2020, a major incident severely impacted database performance and availability at Cloudflare. Cascading failures led to a 75% drop in API availability and 80x slower dashboards. The primary database instances in both main and standby regions executed a failover, but the main region's new primary struggled under the load and failed again. With no synchronous replicas to promote in the cluster, a crucial decision had to be made: promote an asynchronous replica with potential data loss or evacuate regions to the standby cluster, incurring additional downtime.

Data loss is unacceptable, hence the latter option was chosen. Further investigation revealed a partially failed network switch, operating in a degraded state and blocking communication between both cross-region etcd clusters.

The non-communication caused the Raft protocol to enter a deadlock state, causing read-only clusters. Failovers and resynchronization exposed PostgreSQL inefficiencies, particularly in resync times. The setup handles full failures, like machine crashes, effectively, but issues arise with undefined behavior when nodes in your Raft cluster start providing conflicting information.

For example, nodes 1, 2, and 3 experienced degraded network connectivity due to a faulty switch. Node 1 mistakenly thought node 3 was no longer the leader, causing a series of failed leadership elections and creating a deadlock. This deadlock forced the cluster into read-only mode, disrupting communication

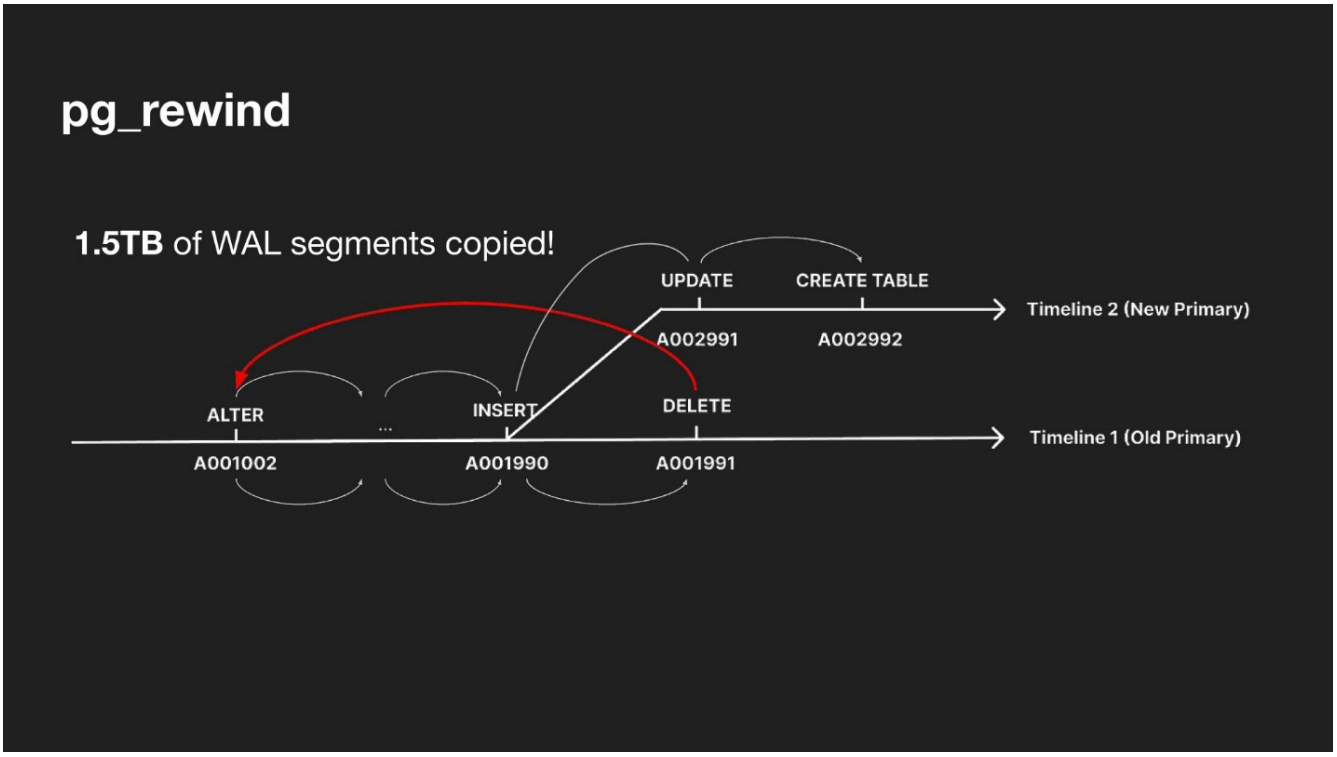


Figure 6: pg_rewind copying all the logs, including the common segments before the last divergence point

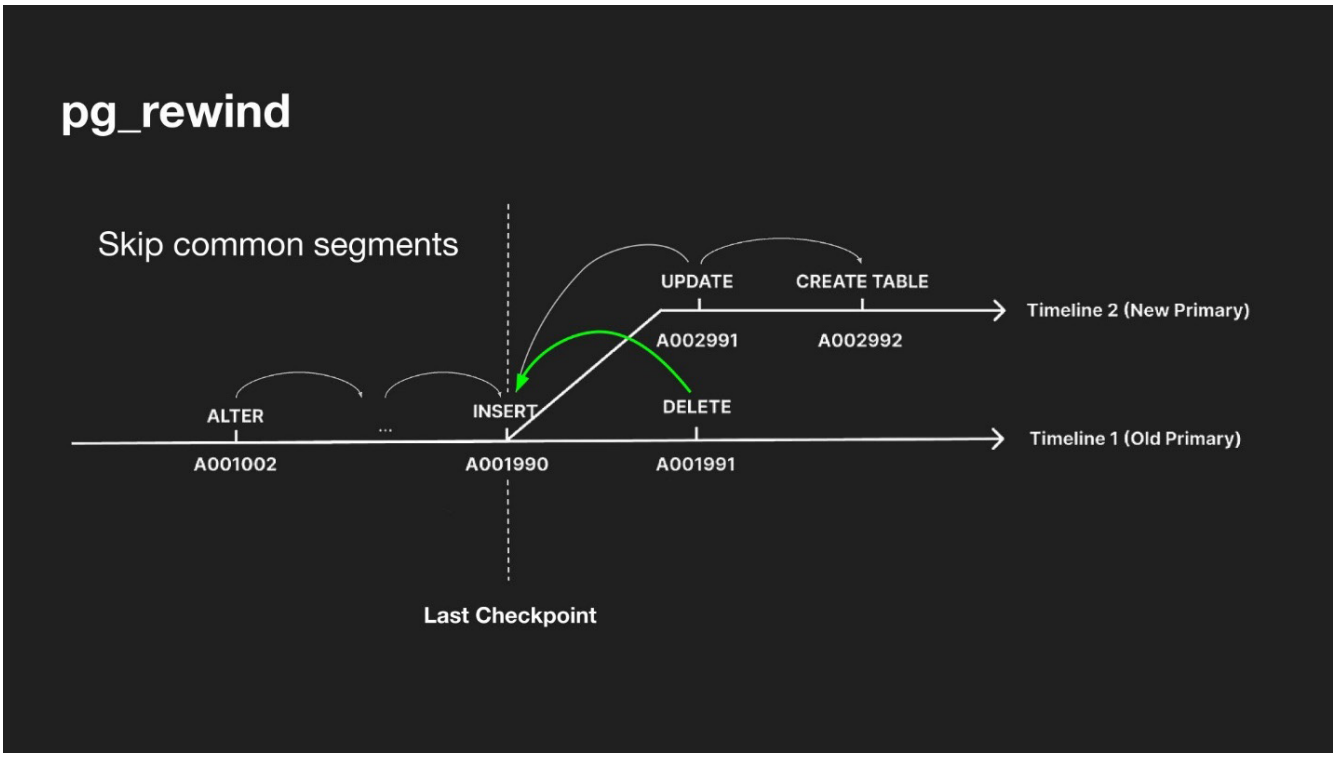


Figure 7: Patched pg_rewind copying logs only from the last divergence point

between regions and triggering a failover to primary replicas.

During failover, the synchronous replica is promoted, requiring the old primary to undo committed transactions due to potential divergence in transaction history. After unwinding the divergent history, the synchronous replica must receive and replay new transactions from the corrected primary, which, in our case, failed due to the absence of synchronous replicas to absorb the new data, resulting in downtime.

The identified issues include a hardware failure, a rare Byzantine Fault with Raft, and prolonged Postgres resynchronization time. The team prioritized optimizing Postgres to address the third issue. Analyzing the logs revealed that most time was spent in rsync, copying over 1.5 terabytes of log files during resynchronization.

The solution involved optimizing Postgres internally by copying only the necessary files from the last divergence point, reducing data transfer to 5% of the original size. Optimizing PostgreSQL internally reduced replica rebuild times from 2+ hours to 5 minutes, a 95% speedup.

The lessons learned from the large-scale incident include:

- anticipating degraded states and not just fully failed states
- investing in open-source contributions for in-house expertise. The open-source patch submitted contributes to PostgreSQL CommitFest.
- the importance of building and fixing software internally.

Access Data from the Edge

Maintaining a single cluster in one region isn't sufficient. To address various failures, including challenging scenarios like Byzantine hardware failure, you need to distribute PostgreSQL clusters across multiple regions. This cluster-of-clusters approach, seen from a Cloudflare Workers' standpoint, enhances resilience. Cloudflare expanded from a US-based primary region to include

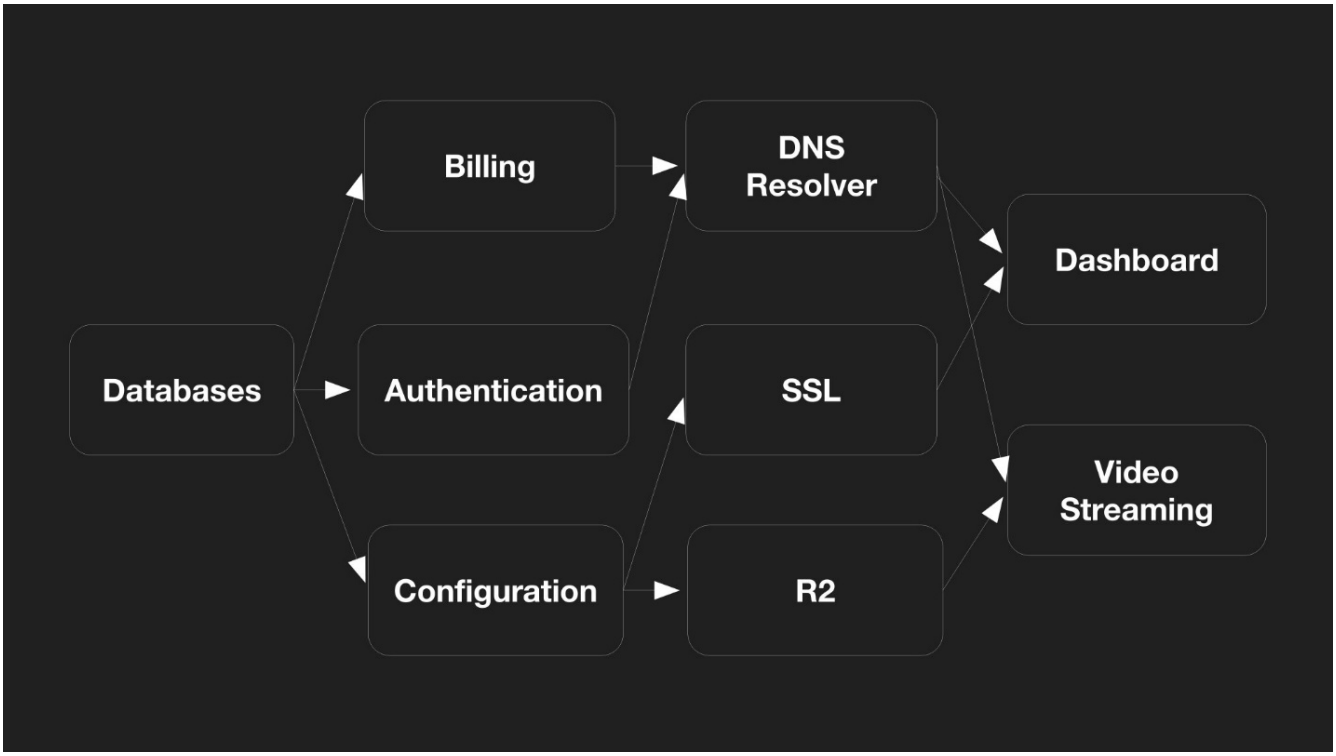


Figure 8: The full application dependency graph

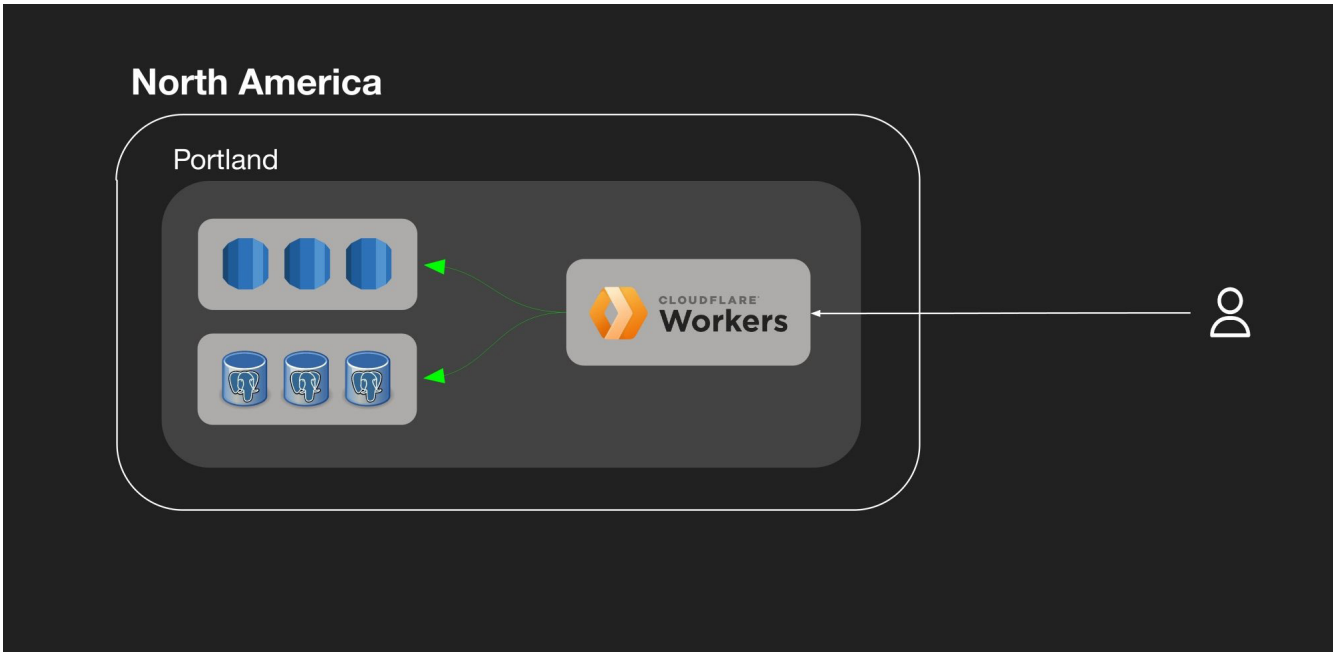


Figure 9: Cloudflare Workers colocated with the database cluster

Europe and Asia, forming a hub-and-spoke model using PostgreSQL streaming replication.

Ensuring synchronization across regions is crucial, handling issues like replication lag:: when faced with high replication lag, diverting traffic back to the primary region helps meet SLAs for application teams. This strategy enhances system robustness and mitigates the impact of potential failures.

A distributed cluster offers significant benefits, particularly in implementing smart failovers. By strategically choosing the primary region based on factors like current load, available capacity, or time zones ("follow the sun"), we optimize for lower latency during active hours in different parts of the world. This dynamic approach addresses PostgreSQL's single-primary limitation.

Capacity considerations also guide the choices: certain regions may have more capacity, and logistical challenges, like hardware shipments during COVID, impact decisions. Traffic patterns and compliance requirements further influence region selection, allowing the team to address specific regulatory needs efficiently. Adopting a distributed approach enables Cloudflare to navigate these challenges effectively.

Expanding PostgreSQL across multiple regions and achieving quick failovers, as discussed earlier with tools like pg_rewind, is efficient. However, the complexity arises when considering application dependencies. While database migration is manageable, failing over entire application ecosystems, including distributed components like Kafka message queues, is a challenge. Understanding the full application dependency graph is crucial when switching over regions.

Despite ongoing automation efforts, coordination between teams, across different organizations, becomes critical. The sequence to switch services across regions involves considering database clusters, application services (e.g., authentication and configuration) in Kubernetes clusters, and their reliance on central infrastructure like R2 and SSL.

Database Trends

As practitioners, we have observed key trends in relational databases. First, there's a move towards embedding data at the edge, using monolithic or PostgreSQL in a few regions, complemented by embedded databases like SQLite for real edge processing. It is vital to keep SQLite at the edge syncronized with PostgreSQL data at core/central locations. Furthermore, ensuring SQLite is PostgreSQL wire-compatible is crucial for seamless integration.

Another trend involves persistence at the edge, with Cloudflare Workers processing client-side data. Additionally, there's a shift towards the colocation of storage and compute, illustrated by features like Smart Placement.

This reduces latency, by avoiding cross-region network hops to an otherwise central data storage and addresses the realization that many client applications spend considerable time communicating with databases.

Lastly, data localization, particularly in Europe using PostgreSQL, is a challenge. Logical replication is a focus, offering flexibility in replicating on a row or column level, although it's still in an exploratory phase. The expectation is that logical replication will play a significant role in addressing this challenge.

Understanding Architectures for Multi-Region Data Residency

by **Alex Strachan**, Staff Software Engineer |Performance & Architecture @Rippling

The main focus of this article is the effective implementation of data residency strategies while ensuring a positive experience for all stakeholders. The central idea revolves around the necessity of placing data in locations beyond a single source. There are various motivations for adopting this approach, such as disaster recovery and geo-redundancy. The core concept is to distribute data, ensuring that specific sets are housed in distinct regions without any shared overlap – a practice referred to as data residency.

First Principles: Know Your Customer (KYC)

The crucial initial step in implementing data residency is identifying the intended beneficiaries and motivations behind the decision. Contrary to common misconception, data residency is not a GDPR requirement. Instead, the driving force is often the specific needs and concerns of your customers. This involves delving into the reasons behind the request for data residency, which may vary widely.

Begin by deciphering who initiated the need for data residency and why. Contrary to popular belief, it might not be directly linked to GDPR compliance. Engage with various stakeholders, including sales, customers, and legal teams, to comprehend the specific project requirements. The goal

is to align the technical implementation with contractual promises regarding data residency.

At Rippling, for instance, the impetus behind this endeavour was customer requests from the EU seeking assurance around data privacy and regulatory exposure. This drove a set of restrictions, including the prohibition of inter-regional communication, highlighting the importance of tailoring the solution to the unique demands of your clientele.

Furthermore, consider scalability as a driving factor. While cell-based architectures can enhance system scalability, the decision to implement fully separated data residency should be rooted in specific needs. Separation and partitioning of data can be advantageous for scaling, and creating independent databases and architectures.

Additionally, data residency can contribute to your compliance story. For organizations dealing with multiple governments, each with distinct regulations and compliance requirements, separating data by region becomes very useful. For example, compliance jobs related to specific government regulations, such as the UK's unique labour laws, can be efficiently managed when the data resides in the same region. This not only addresses potential data access issues but also ensures alignment with varying compliance needs across different regions.

In essence, the key takeaway is to understand the intricacies of your organization's requirements and motivations. Avoid a one-size-fits-all approach, and instead tailor the data residency strategy to the specific needs of your company, stakeholders, and the regulatory landscape you operate within.

Truth and Trust

A critical principle in the context of multi-region deployments is establishing clarity on truth and trust. While knowing the source of truth for a piece of data is universally important, it becomes especially crucial in multi-region scenarios. Begin by identifying a fundamental unit, an "atom," within which all related data resides in one region. This could be an organizational entity like a company, a team, or an organization, depending on your business structure.

Any operation that involves crossing these atomic boundaries inherently becomes a cross-region scenario. Therefore, defining this atomic unit is essential in determining the source of truth for your multi-region deployment.

In terms of trust, as different regions hold distinct data, communication between them becomes necessary. This could involve scenarios like sharing authentication tokens across regions. The level of trust between regions is a decision rooted in the specific needs and context of your business. Consider the geopolitical landscape if governments are involved, especially if cells are placed in regions with potentially conflicting interests.

Before embarking on multi-region deployments, conduct a thorough threat model analysis in collaboration with your security team or consultants. Understand the potential risks and regulatory requirements in each country involved. Two crucial aspects that demand attention are access tokens and database access, as every application in this context must address these concerns.

The trust between regions can be reduced through cryptographic techniques, but the specific approach will depend on your business requirements. For example, you may need to evaluate the trade-offs between stateless communication and revocation delays based on your application's needs.

Initiate conversations with your team focusing on these two aspects – access tokens and database access – as they often lead to discussions around other pertinent considerations for your business. Establishing a robust philosophy and architectural foundation before implementation is crucial to avoid reevaluating fundamental design patterns later, especially when expanding into new markets or regions.

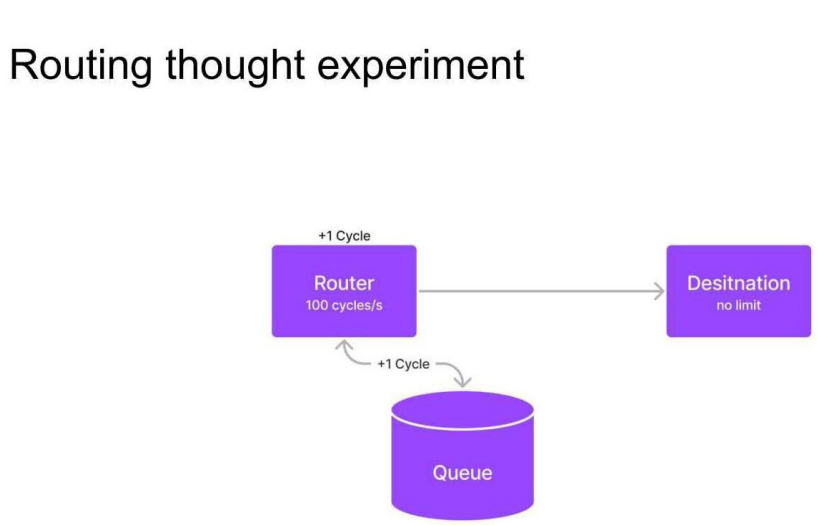


Figure 1: Routing thought experiment

Do the Same Thing Every Time

The principle of consistently doing the same thing applies universally to software engineering but becomes particularly vital in multi-region scenarios. Avoiding unnecessary forks in the design path is crucial for robustness and reliability. This concept can be illustrated by revisiting a historical example related to network routing.

Consider a router design where, in regular routing mode, the system operates efficiently. However, if the load exceeds a certain threshold, a queue is introduced. This seemingly reasonable design can lead to a catastrophic failure if the system operates at over 50% capacity. A single traffic spike can result in perpetual network downtime due to the inherent inefficiency of the design under such conditions.

Applying this to multi-region scenarios, where routing decisions are frequent, it's essential to opt for designs where the same action is taken consistently. For instance, routing incoming traffic, avoid designs that include logical forks like "if multiregion; then". Doing the same thing every time simplifies the system, making it more predictable and easier to maintain.

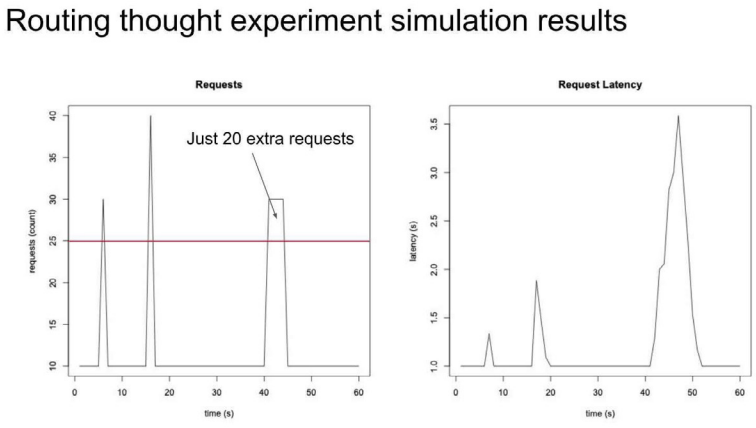


Figure 2: Simulation results

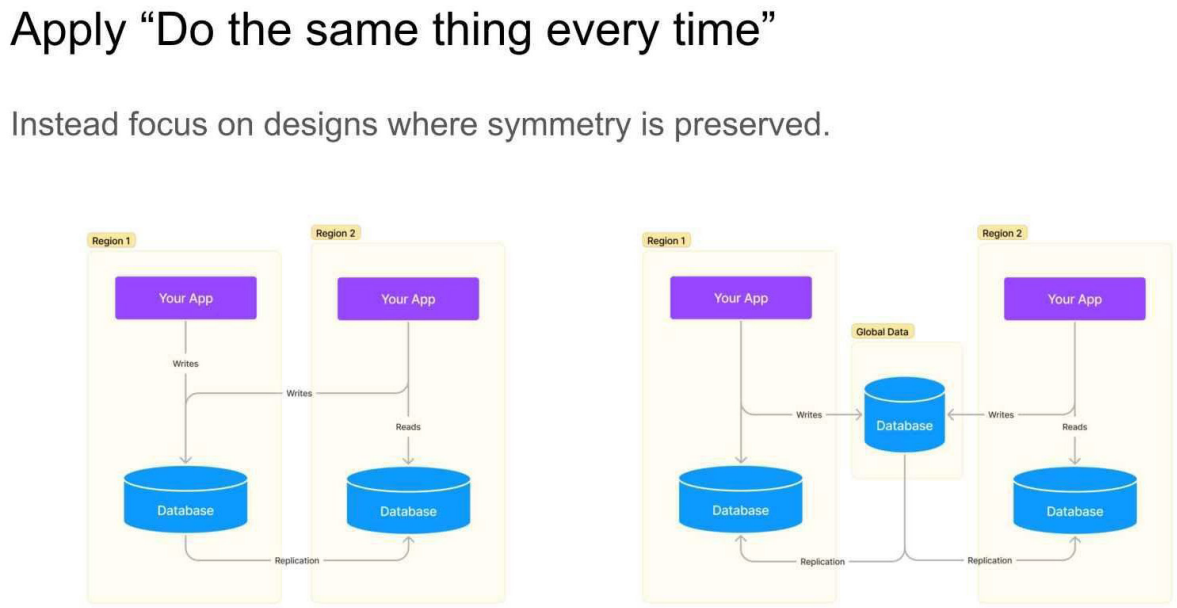


Figure 3: Apply “Do the same thing every time”

Moreover, minimizing complexity and reducing the number of branches in the system enhances the likelihood of success. The addition of features, especially those with low user percentages, increases the chances of encountering unforeseen issues. The probability of a feature working is directly proportional to the fraction of users using it divided by the complexity of the feature.

Specifically, in multi-region scenarios, caution is advised when considering geographic proximity for routing decisions. Implementing rare code paths for exceptional circumstances, like diverting to another region, can be brittle and challenging to test thoroughly. Asymmetry in the application’s behaviour between regions, especially concerning synchronous consistency, can lead to subtle bugs that are hard to track down.

To illustrate, consider scenarios where one region writes synchronously to its local database, while another region writes asynchronously to a remote database. Reading just-written data in the first region will give immediately correct results where in the second region you would read stale data. Such inconsistencies can result in unpredictable behaviour and should be avoided. Striving for symmetry in designs, where both regions share similar code paths, ensures a more straightforward and reliable system. While this might introduce some complexity, the benefits of uniformity across regions outweigh the potential downsides.

Designs: Where to draw the Circles

In a cell-based architecture, where your application is likened to an onion with distinct layers (edge, app servers, and database), implementing effective client routing becomes pivotal. The objective is to encapsulate the inner contents of each layer as a single unit visible to external entities.

Client Routing

For client routing, a straightforward approach is to allocate subdomains for each region, such as eu1.yourapp.com and us1.

yourapp.com. This method simplifies the client's role in routing decisions and has been an initial solution employed by early adopters of cell-based architectures. While effective, it may inconvenience customers who need to manage routing logic themselves, particularly when their domain of concern aligns with the designated atom for data residency.

Despite its simplicity, this approach presents challenges when customers need to switch regions. If a user submits a request to change from Region 1 to Region 2, all their integrations relying on specific URLs and assumptions tied to the client domain knowledge of routing will break. While there are cases where this strategy is necessary, especially in data residency shifts, it might not be the optimal first-choice strategy.

Consider alternatives that offer a smoother experience for customers and minimize disruption during region transitions. Prioritize strategies that provide flexibility and ease of adaptation, reducing the burden on customers when navigating changes in the underlying infrastructure.

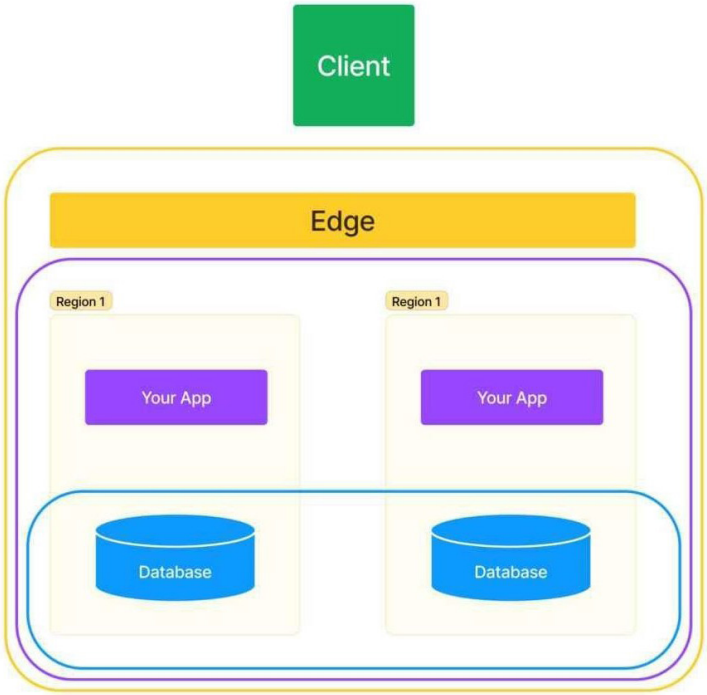


Figure 4: Application Onion

Gateway Routing

In the context of a cell-based architecture, where atoms represent indivisible components of an application, introducing gateway routing can offer an efficient approach to client requests. The idea involves embedding the atom ID in the request and leveraging a mapping mechanism at the edge to determine the appropriate region.

For traffic residing within the atom boundary, this method is highly effective. By including a unique identifier, such as the company ID, in the request (as a header, query parameter, or path parameter), it serves as the routing key. The mapping at the edge then ensures that the traffic is directed to the correct region. If a region change occurs, updating the mapping at the edge suffices to reroute the traffic seamlessly.

However, this approach encounters challenges when dealing with cross-atom traffic. In scenarios where a partner needs to access multiple companies, each operating in different regions with distinct functionalities, the traditional gateway routing method falls short. Simply fanning out to the partner's first region and performing operations won't suffice.

Addressing cross-atom traffic requires additional considerations and strategies. It's unlikely that gateway routing will be sufficient to address all use cases but it's an effective approach for handling the large majority of traffic volumes in most applications.

Application Routing

An effective strategy to address cross-atom traffic is allowing communication between regions. By enabling regions to communicate with each other, you create a framework where they can make calls, facilitating data access and interactions across regions. The feasibility of this approach hinges on the level of trust established between the regions, governed by your security policies and compliance requirements.

If your security policies permit cross-region data access for legitimate business purposes, which is often the case, enabling inter-region communication becomes a valuable tool. For instance, if your application requires a query that spans multiple companies, having regions capable of cross-talk can be advantageous.

Utilizing the atoms as a routing key, you can implement a batch API for efficient handling of queries involving multiple companies. This way, you can automate the routing to the appropriate regions based on the list of company IDs involved. While leveraging atoms is preferred when feasible, there might be instances, such as when conducting audit jobs across all regions, where a service operates independently of atoms.

However, it's crucial to note that allowing cross-region communication introduces complexity and may result in increased latency due to the geographical distance between regions. Regional latency, especially in cases like New York to Shanghai with a potential round trip time of 400 milliseconds, needs careful consideration. It's essential to assess the trade-offs and structure your application to minimize the impact of inter-region calls, perhaps by collecting data requirements upfront and optimizing the flow of requests to avoid unnecessary and latency-intensive calls.

Database-Managed Locality

There are a lot of databases out there that will say, "we're a global database". They often are. They're cool. They do stuff like replication. You can give a region key and it'll store the data in that region. We do have to talk through that a little bit because you have to think through what kind of use case they were trying to enable to decide whether this magic database is going to work for you.

Database-Managed Locality: Accelerators

- Adopt an active-passive topology, reminiscent of CDN functionality.
- Involve writing data once, with all regions receiving an eventually consistent copy.
- Ideal for accelerating local data flow, but not inherently tailored for data residency due to the data-copying nature.

Despite not being the core solution for data residency, accelerators excel in scenarios involving universal, consistent, and infrequently changing data. The U.S. tax code serves as a prime example, showcasing the effectiveness of replicating such data cross-region using an accelerator-type pattern.

Database-Managed Locality: Global Databases

- Embrace a true global presence with active-active topology across regions.
- Allow both write and read operations from any location, guaranteeing eventual consistency.

While true global databases provide powerful capabilities, challenges arise in scenarios where an active-active topology follows a last-write-wins approach. This can lead to conflicts when multiple regions act on the same object simultaneously, requiring careful consideration of conflict resolution methods beyond a simple last-write-wins strategy.

Synchronous conflict resolution may necessitate a quorum between all nodes, incurring significant cross-region latency. Additionally, partial replication, where specific data sets are chosen for replication, is not universally supported and requires a thorough examination of database specifications.

In the pursuit of data residency solutions, relying solely on placing a database proxy in front is not a one-size-fits-all remedy. A nuanced understanding of technology capabilities, careful examination of specifications, and thoughtful consideration of conflict resolution methods are essential components of a successful data residency strategy.

Summary

Pulling it all together, you are going to have to ask your team who it is you're building this for, and what exactly the requirements are. It may surprise you, the answers aren't one-size-fits all and will likely differ from your expectations. This is an incredibly important part of the exercise. When you're working through trust and truth, it's important to define an indivisible part of your architecture and then an encapsulating idea, so things above can be cross-region and things below never are cross-region. That will help immensely in keeping yourself sane when developing code.

Do the same thing every time. When you see forks in the road, especially related to region resolution related to business logic in that domain, I would strongly encourage you to think of ways to try to work around those forks in the road so that the code path followed is consistent. Again, when you're testing, when you're building your test and rollout plan when you're making changes in the future, it's going to be hard to know that you've broken one of these delicate mechanisms that are on a rare code path, try to get rid of those.

Managing 238M Memberships at Netflix

by **Surabhi Diwan**, Senior Software Engineer @Netflix

Surabhi Diwan, a Senior Software Engineer at Netflix, presented at QCon San Francisco 2023 on Managing 238M Memberships at Netflix. In her talk, she shared how the Netflix membership team does distributed systems: the architecture bets, technology choices, and operational semantics that serve the needs of Netflix's ever-growing member base.

She started with some of the pricing and technology choices Netflix made probably a decade ago that predate her tenure at Netflix. She then moved on to a use case study for member history, which is the second persistent store that helps us answer questions about the microsecond granularity of any and every single change made to anybody's subscription.

"I'm sure most of you are Netflix members. If you're not, I will show you exactly how to sign up as we get into the details of this. Finally, I will try to answer the question: What does the subscription ecosystem evolution look like? It has 238 million subscribers. Really, what is the journey like? What does that look like if you have to add another 5 million subscribers? If you have to add another 100, what does that look like?"

Membership Engineering

The team's primary focus with memberships lies in the critical pathways of signups and streaming at Netflix. They manage a suite of various microservices, totaling a dozen within the membership domain. With a commitment to four nines availability, their mid-tier services ensure uninterrupted user access, directly impacting signup processes and streaming experiences.

These services handle substantial traffic, scaling up to millions of requests per second depending on the specific use case, such as subscriptions or pricing. The team discusses their technical decisions, leveraging off-the-shelf technology to achieve scalability and reliability. They serve as the authoritative source for global membership counts, facilitating downstream analytics both within and outside Netflix. In essence, they tackle complex distributed challenges at scale with precision.

Additionally, the team oversees the catalog of Netflix plans and pricing, which, despite its simplicity - basic, standard, premium - plays a crucial role in user experience. They prioritize accuracy in subscription details, ensuring correct plan selection, billing country, and payment status to maintain service quality and member satisfaction.

Managing the entire membership lifecycle, they facilitate signups, partner channel integrations, plan changes, renewals, and account closures. Their responsibilities include handling payment issues, including account holds and cancellations, and ensuring compliance with data privacy regulations by appropriately managing customer data throughout the membership journey.

When Do Members Use Our Flows?

I think that was a broad establishment of what my team does. As end users, I think you would care if you were to go on the Netflix app right now. These are some of the places.

The flow highlights critical user interactions, such as the start membership and play buttons, which trigger backend processes managed by the Netflix Membership engineering team. A flowchart illustrates the various services ensuring a seamless membership experience before users can begin streaming. The play button directly interacts with membership systems to determine service quality based on the user's plan. This interaction generates the highest traffic volume due to the billions of streams initiated daily. Additionally, user actions on the account page, such as plan changes or managing additional members, are facilitated by membership services. Partner signups, exemplified by Xfinity activations, are also orchestrated by the team's backend services.

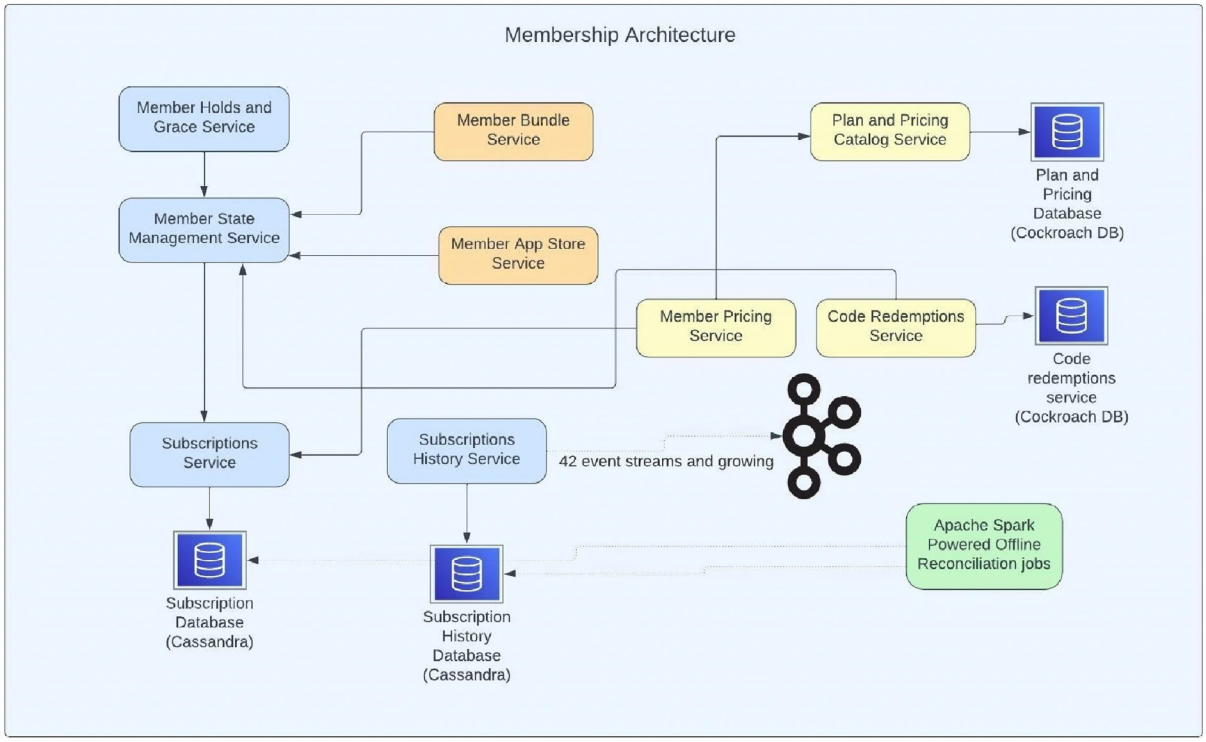
How Do We Make It Happen?

I think this is the meat of the puzzle: establishing what we do. This is really how we do it. It is a little hard to unpack.

The team manages the membership plan and pricing catalog, storing and managing plans globally, with variations across different regions. This service also handles rule management based on location-specific offerings. They utilize two CockroachDB databases to support plan pricing and code redemptions, particularly during peak gifting seasons. The member pricing service facilitates member actions, such as plan changes and adding extra members.

Partner interactions are managed by a dedicated microservice that handles bundle activations and signups, including integration with platforms like Apple's App Store for subscription signups. Membership data is stored in Cassandra databases, supporting subscription services and history tracking for over 238 million active memberships.

The team's focus extends beyond current members, including former members and rejoin experiences. They manage states with services like member states and member holds, ensuring smooth operations and reconciliation across databases using tools like Casspactor and Apache Spark for big data processing. This data, such as messaging and analyt-



ics, is crucial for downstream consumers to derive insights on signups and revenue projections.

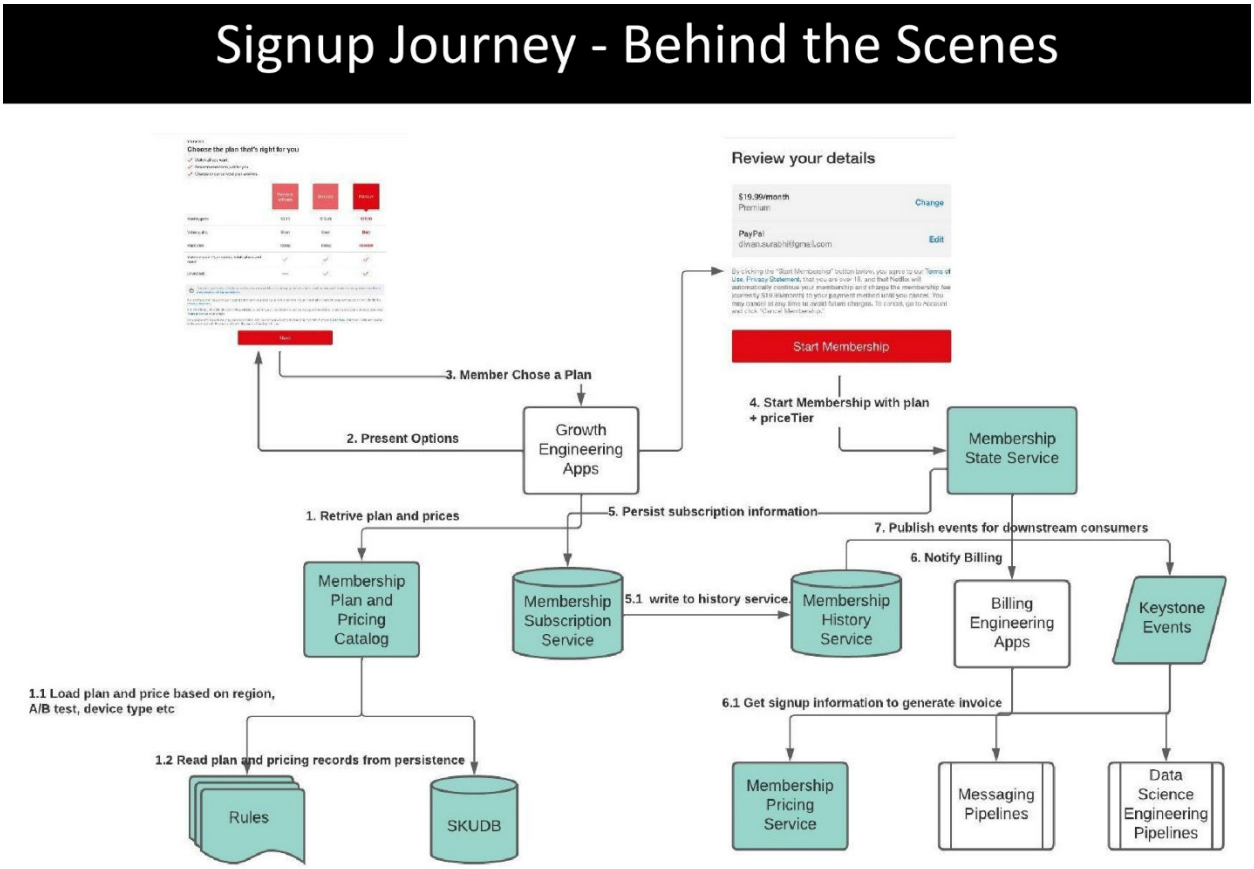
Signup Journey

Once users begin their journey on Netflix, they encounter plan selection options powered by membership systems, which handle millions of requests per second. Rendering this page correctly is crucial due to geographical variations in currency, pricing, and available plans. The team manages these rules, with green boxes representing membership services' responsibilities and white boxes representing collaboration with sister teams.

The process starts with plan selection through growth engineering apps. The chosen plan is queried from the membership plan and pricing service, backed by CockroachDB for plan pricing details. After confirmation, hitting "start membership" triggers actions in the membership state and history services, updating persistent records with relevant information like plan, price tier, and country. Events are emitted to signal the membership's activation, triggering messaging pipelines for welcome emails and informing downstream teams for analytics. Despite the simplified explanation, this process operates at a large scale within a distributed system, requiring robust error handling and reconciliation.

Membership Team Tech Footprint

Netflix operates on a distributed systems architecture optimized for high read requests per second (RPS). With over 12 microservices, they utilize gRPC for communication at the HTTP layer. Their primary coding languages are Java, and they are transitioning to Kotlin for app rewrites, all brought together with Spring Boot. Kafka plays a significant role in message passing and interfacing with other teams, such as messaging and downstream analytics. Additionally, Netflix employs Spark and Flink for offline reconciliation tasks on their big data, which we›ll explore in more detail later.



Technology Choices for Operations and Monitoring

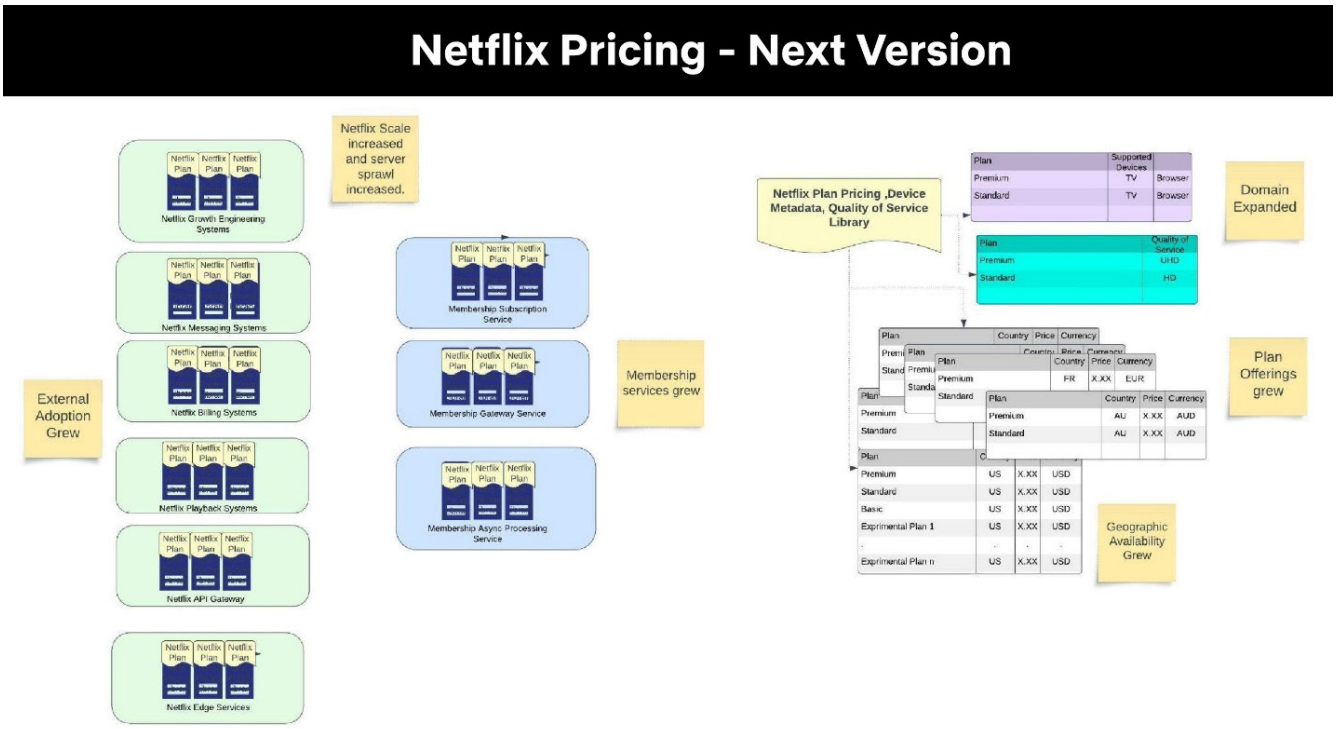
Beyond coding and deploying to production, the Netflix Membership engineering team takes on-call responsibilities, addressing critical failures promptly. They employ lightweight transactions and try to ensure data consistency in our online systems by utilizing tools like Cassandra. Reconciliation jobs, powered by Spark and Kafka, ensure alignment between systems of record within the membership, such as subscriptions and member history databases. This accuracy extends to external systems, maintaining a consistent state across the ecosystem. Data alerts and repair jobs monitor and rectify discrepancies, ensuring every record reflects the latest information.

In observability, extensive logging, dashboards, and distributed tracing facilitate rapid error detection and resolution, which is crucial in Netflix's vast microservices landscape. Production alerts track operational metrics, guaranteeing optimal service levels. Operational data also fuels machine learning models for enhanced anomaly detection and automated issue resolution, maintaining uninterrupted streaming experiences for our members.

Use Case Studies

So far, we have established the Netflix Membership engineering team's position, architecture, and core operational flows. It's vital to delve deeper into potential improvements and future readiness. Comparing system design to chess, mastering it requires understanding rules and strategies and analyzing past moves for improvement.

We will reflect on Diwan's five-year tenure; the team examined past architectural decisions, mainly focusing on Netflix's pricing, technology choices, and the importance of cataloging system modifications. In addition, we ex-



plore their journey in scaling, from initial requirements to continuous evolution and improvement.

Learning from the Past - Netflix Pricing Tech Choices

A decade ago, Netflix's pricing architecture was simplistic, managing only a few plans and basic functionality. Initially, a lightweight, in-memory library served these needs efficiently. However, as Netflix expanded globally and diversified its offerings, this library grew in scope and complexity, becoming vital across multiple applications. Over time, operational challenges emerged due to its size and dependencies, necessitating a transition to a more robust architecture.

The new plan pricing architecture utilizes CockroachDB for persistence and a gRPC service to manage traffic. Despite the simplified design, migrating from the legacy library was a multi-year endeavor involving numerous engineering teams and applications. This highlights the importance of future-proofing architectural decisions and promptly addressing tech debt to avoid costly issues.

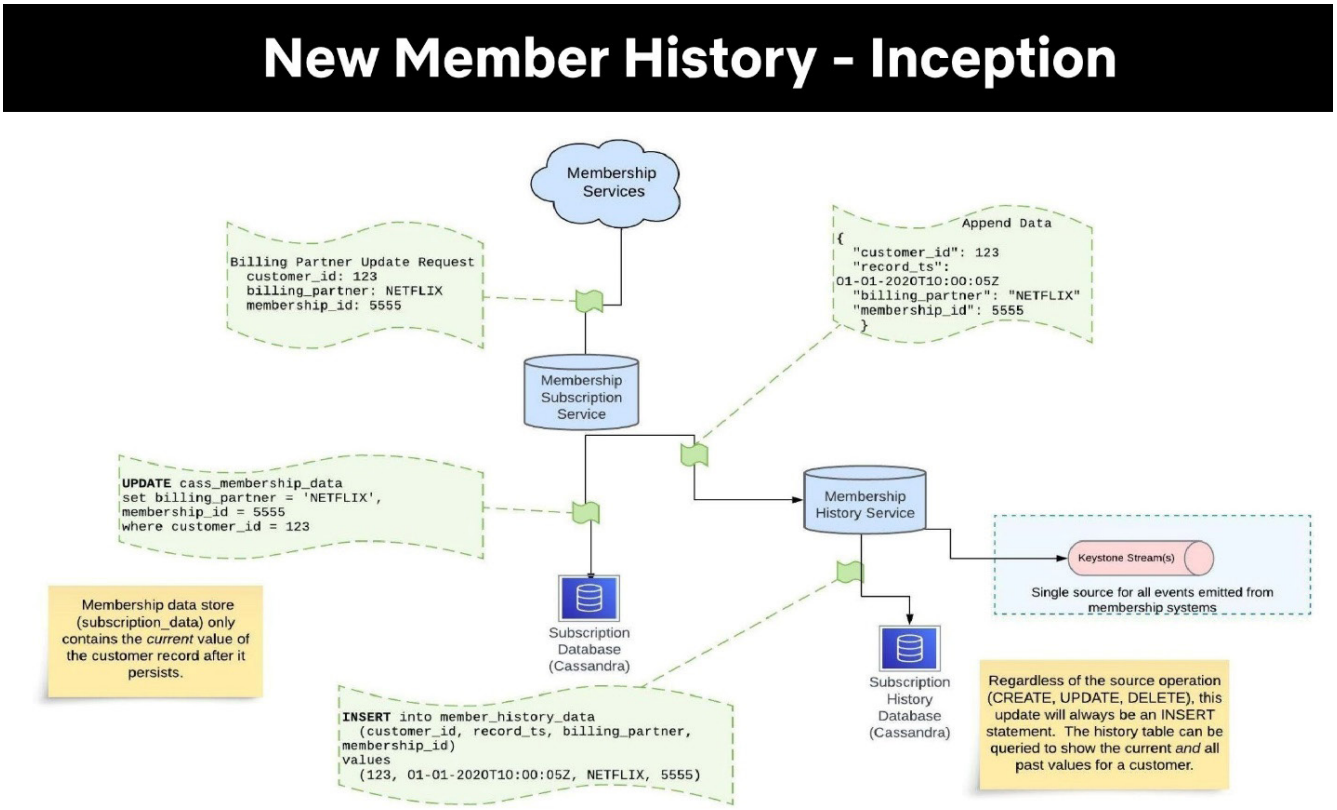
While the new architecture is the primary solution, remnants of the old library still linger, requiring ongoing migration efforts. This underscores the need to consider long-term implications and proactively address legacy systems during technology transitions.

Member History

The study on member history dives deep into its evolution and pivotal role within Netflix's architecture. Initially, member history was tracked through application-level events, but the need for granular data persisted. As Netflix expanded globally, the complexity and criticality of member data grew, necessitating a more robust solution.

The new architecture employs a change data capture pattern, recording direct delta changes on operational data sources. This append-only log system, backed by a Cassandra database, offers comprehensive tracking of membership events. By centralizing event emissions through member history streams, Netflix gained improved observability and the ability to reconcile data across systems.

The benefits of this architecture are manifold. It enables detailed debugging, replaying of events, and seamless reconciliation in



case of data corruption. Moreover, member history enriches customer service analysis and feeds into downstream analytics, messaging, and membership data systems.

Despite the multi-year journey to implement this architecture, the dividends have been substantial, underscoring the importance of investing in architectural innovations for long-term success.

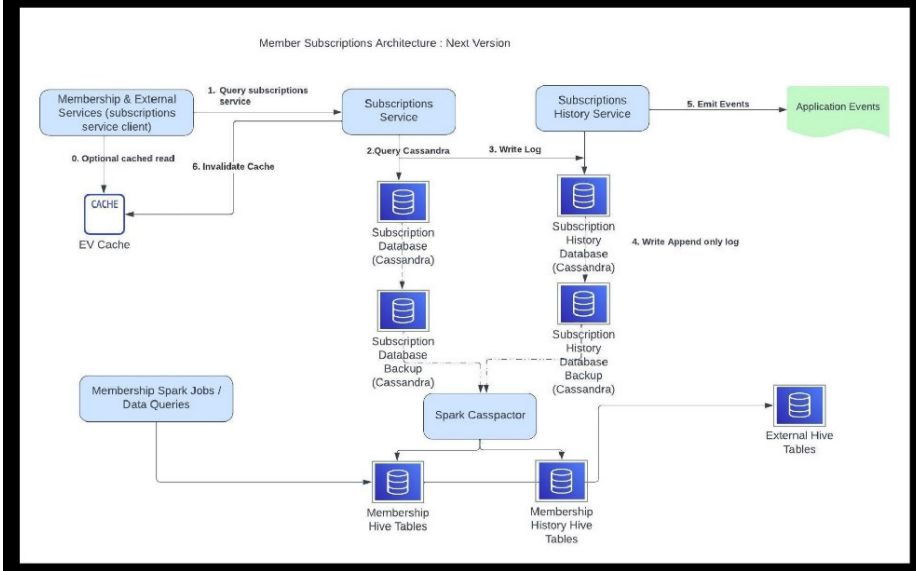
Preparing for the Future - Member Subscriptions Ecosystem Evolution

As we enter the final stretch, let's delve into the evolution of the subscription ecosystem. Initially, we made basic architectural choices and relied on off-the-shelf components like gRPC services and Cassandra databases for scalability. However, as our user base grew, we encountered challenges reconciling data and ensuring fault tolerance.

To address these issues, we implemented a Spark Casspactor to manage backups and reconcile data in Hive tables, enabling better auditing and self-healing. While this improved debuggability and removed single points of failure, scalability remained a concern. To mitigate this, we're considering caching with EVCache for faster lookups, albeit with some trade-offs in consistency.

The key lesson here is that no system can infinitely scale, and it's crucial to continually invest in innovation and architectural evolution to stay ahead of system limitations and avoid unexpected outages.

Member Subscriptions Architecture - Current and Evolving



- When consensus based lookups will not serve us well; hash based lookups will.
- Fronting our persistence store with cache to keep our latency low irrespective of data footprint.
- Trading off for performance at the cost of lower consistency

Conclusion

The pivotal lesson from Netflix's pricing decision is the importance of future-proofing its technology choices and being proactive in adapting or pivoting when necessary. Similarly, the member history case underscores the potential payoff of bold architectural investments. It's essential to have the courage to pursue significant innovations.

The evolution of member subscriptions is a continuous process with no definitive endpoint. This ongoing challenge reminds Diwan of a famous quote in computer science:

There are only two hard things in computer science: cache invalidation and naming.

While Diwan focuses on caching, she recommends watching "The Queen's Gambit" for a captivating story about pursuing excellence.

Unpacking How Ads Ranking Works at Pinterest

by **Anthony Alford**, Director, Development at Genesys Cloud Services

Aayush Mudgal, Staff Machine Learning Engineer at Pinterest, presented at QCon San Francisco 2023 a session on Unpacking how Ads Ranking Works at Pinterest. In it he walked through how Pinterest uses deep learning and big data to tailor relevant advertisements to their users.

As with most online platforms, personalized experience is at the heart of Pinterest. This personalized experience is powered through a variety of different machine learning (ML) applications. Each of them is trying to learn complex web patterns from large-scale data collected by the platform.

In his talk, Mudgal focused on one part of the experience: serving advertisements. He discussed in detail how Machine Learning is used to serve ads at large scale. He then went over ads marketplaces and the ad delivery funnel and talked about the typical parts of the ad serving architecture, and went into two of the main problems: ads retrieval and ranking. Finally, he discussed how to monitor the system health during model training and wrapped up with some of the challenges and solutions for large model serving.

Content Recommendation

Mudgal first presented the characteristics of a content recommendation system. Every social

media platform has millions or billions of content items that it could potentially show to users. The goal is to find items that are relevant to a particular user, but since the content catalog and user base are so large, a platform like Pinterest cannot precompute the relevance probability of each content item for each user.

Instead, the platform needs a system that can predict this probability quickly: within hundreds of milliseconds. It must also handle high queries-per-second (QPS). Finally, it needs to be responsive to users' changing interests over time. To capture all of these nuances, platforms need to make sure that the recommendation system solves a multi-objective optimization problem.

When a user interacts with a particular element on a platform, they are often presented with a variety of similar content. This is a crucial moment where targeted advertisements can come into play. These ads aim to bridge the gap between users' and advertisers' content within the platform. The goal is to engage users with relevant content which can potentially lead them from the platform to the advertiser's website.

This is a two-sided marketplace. Advertising platforms like Pinterest, Meta, Google help to connect users with advertisers and the relevant content. Users visit the platform to engage with the content. Advertisers pay these advertising platforms so that they can show their content so that users engage with it. Platforms want to maximize the value for the users, the advertisers, and the platform.

Advertising Marketplaces

Advertisers want to have their content shown to the users. It could be as simple as creating an awareness for that brand, or driving more clicks on-site on the platform. When they do this, the advertisers can also choose how much they value a particular ad shown on the platform.

Advertisers have the option to select from two main bidding strategies. One approach allows advertisers to pay a predetermined amount for each impression or interaction generated via the platform. Alternatively, they can set a defined budget and rely on the platform's algorithms to distribute it optimally through automated bidding processes.

Next, the advertisers also choose their creative or image content. Before serving the creative, the advertising platform needs to define what›s a good probability score for deciding to serve this particular content to a user. This could be defined as a click prediction: given a user and the journey they are taking on the platform, what's the probability that this user is going to click on the content?

However, maximizing clicks might not give the best relevance on the platform: it might promote spammy content. Platforms sometimes also have shadow predictions such as «good» clicks, hides, saves, or reposts that are trying to capture the user journey in a holistic way. On some platforms, there may be more advertising objectives like conversion optimization, which is trying to drive more sales on the advertiser›s website; this is challenging to capture, as conversion happens off the platform.

Also, suppose the platform wants to expand the system to more content types, like videos and collections. Not only do they need to make these predictions that are shown here, but they also need to understand what a good video view is on the platform.

Finally, the different platform surfaces also have different contexts. This could be a user's home feed, where the platform doesn't have any context or relevance information at that particular time, or a search query where the user has an intent behind it.

Given this complexity, as the platform scales it needs to make sure that it is able to make all these predictions in a performant way. Some of the design decisions that are taken here also cater to support scaling and product growth.

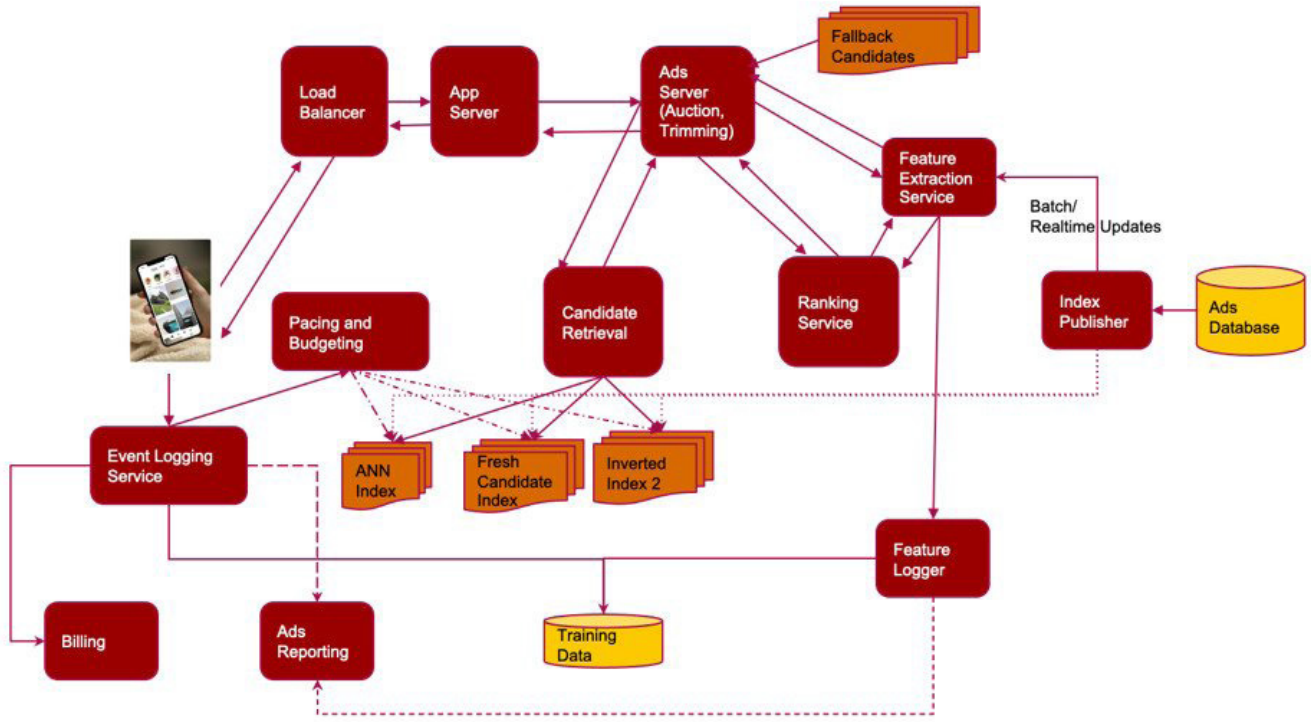


Figure 1: Ads Serving Infrastructure High Level Overview

Ads Serving Infrastructure

Mudgal then presented a high-level overview of the ads serving infrastructure at Pinterest. When a user interacts with the platform, the platform needs to fetch content that it wants to show to the user. The user's request is passed in via a load balancer to an app server. This is then passed to an ad server which returns ads that are inserted into the user's feed.

The ad server needs to do this in a very low latency manner, around hundreds of milliseconds, end-to-end. The input to the ad server is typically rather sparse: a user ID, the user's ip address, and time of the day, for example.

The first task is to retrieve features for this user. This could be things like the user's location from their IP address, or how this user has interacted on the platform in the past. These are usually retrieved from a key-value store where the key is the user ID and the values are the features.

Once this system has enriched the feature space, these are then passed into a candidate retrieval phase, which is trying to sift through billions of content items trying to find the best set of candidates to find hundreds or thousands of candidates which could be shown to the user. Then these are passed into a ranking service, which uses heavyweight models to determine the user's probability of interaction with the content across multiple objectives (click, good clicks, save, reposts, hides).

This ranking service also typically has access to feature extraction, since the system cannot transmit all the content features in a candidate ranking request performantly. Typically, hundreds to thousands of candidates are sent into the ranking service, and sending all of those features together would bloat the request.

Instead, these features are fetched through a local in-memory cache (which could be something like leveldb), and to ensure

maximized cache hits, could utilize an external routing layer. Finally, the ranking service then sends the ads back to the ad server.

In most traditional machine learning systems, the values of the features that are used to show that ad through a particular time are very important to train machine learning models. In addition to the synchronous request to fetch these features, there is also an asynchronous request that's sent to a feature logging service which logs them. Also, to make the system more performant, there are fallback candidates: if any part of the system fails, or is unable to retrieve candidates, fallback candidates can be shown to the user so that the user always sees some content on the platform.

From the ad server, ad content is returned and inserted into the feed for the user. As the user now interacts with the feed, there is an event logging service, which could use Apache Kafka to log all of these events in real-time. This event logging service is very important, since advertisers are billed if the user interacts or clicks on an ad.

Furthermore, advertisers must be billed in real-time, because they define the maximum budget they can spend in a day. If the logging pipeline does not have real-time performance, the platform might overshoot an advertiser's budget or deliver free impressions back to the advertisers.

The event logging pipeline also feeds into a reporting system, which includes hourly or daily monitoring systems. This reporting system also has a linkage to the features that are logged, because platforms want to show advertisers data about ad performance with respect to different features like country, age, or other features that might be on the platform. Finally, the event logging service and the feature logger together combine the training data for all of Pinterest's machine learning models.

Ads Delivery Funnel

Mudgal then showed the ads delivery funnel in more detail. This is broken down into three steps: retrieval, ranking, and auction. In the retrieval step, there are millions of parallel-running candidate generators: given a request, their motivation is to get the best set of ad candidates. This could be based on several criteria, like fresh content, a user's recent interactions, or embedding-based generators. The candidates are then passed into a ranking model, which is trying to predict for different engagement predictions discussed earlier.

Given those predictions, the auction step determines the value of serving a particular ad to the user in the overall context. Depending on the value of that ad, the platform can decide to show it to the user or not. Also, different busi-

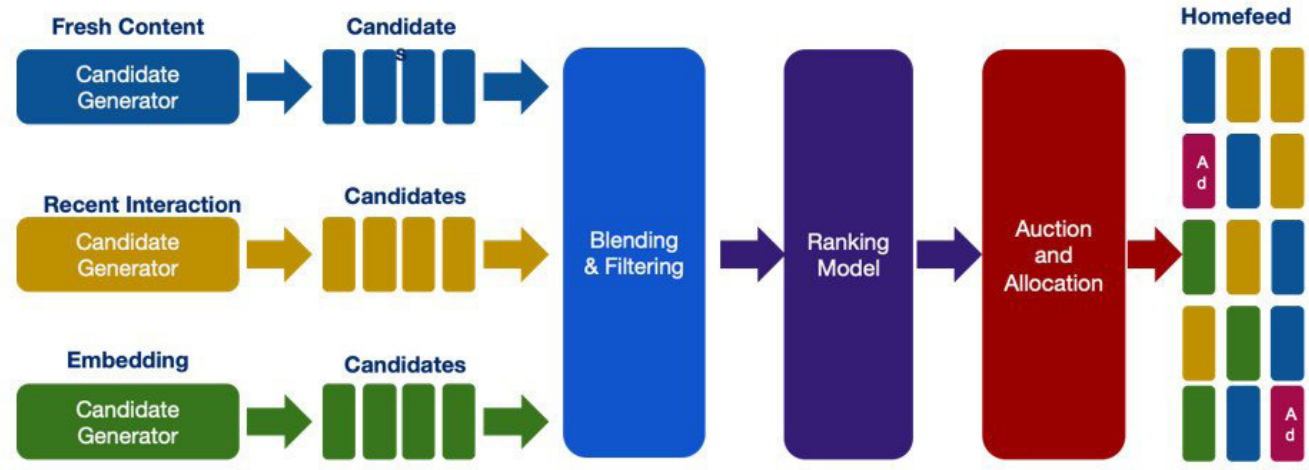


Figure 2: Ads Delivery Funnel

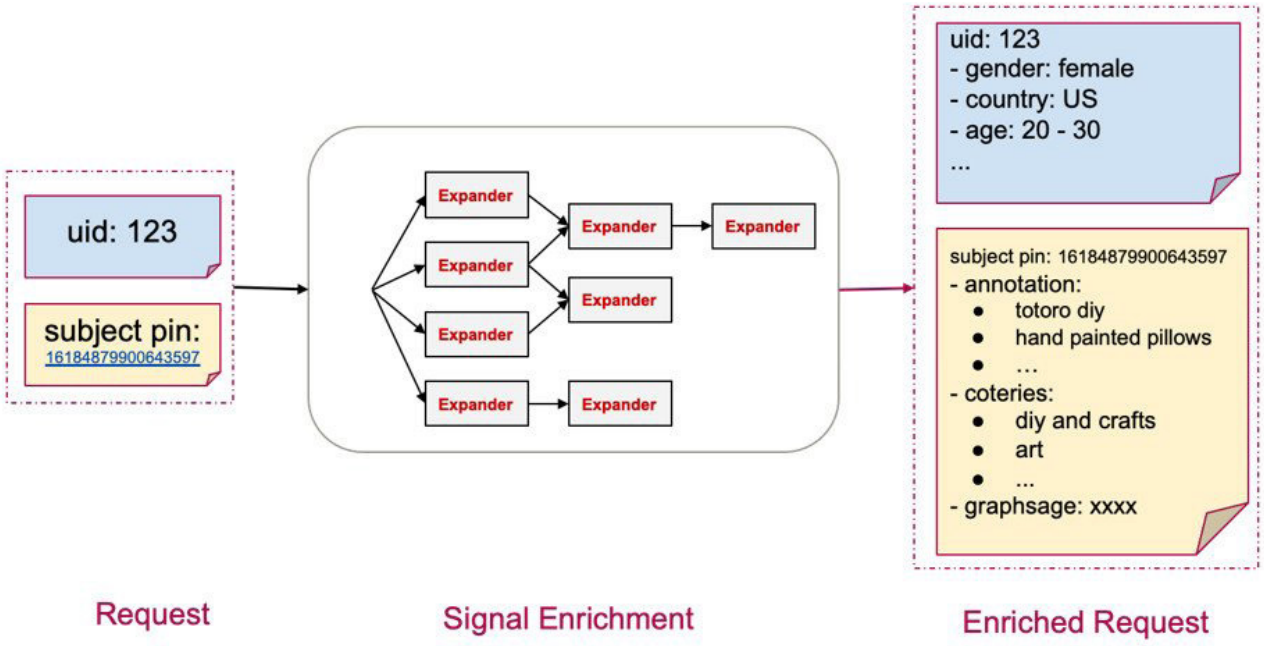


Figure 3: Signal Enrichment

ness logic and constraints around allocation can be handled at this time: for example, should the platform keep two ads together in the feed, or separate them?

Ads Retrieval

The main motivation of ad retrieval is to select the best ads candidate with the best efficiency. This process uses very lightweight ML models, which can run at a very low computing cost. The quality metric for the models is recall.

Remember that the input to this system is the user's ID and the content ID and request-level features. Retrieval requires signal enrichment, which uses several graph-based expanders to fetch extra features from key-value feature stores. For example: a user ID maps features such as age, location, gender, prior engagement rates. Similarly, the content ID maps to content features that are precomputed in the pipeline to reduce computation and improve online latency.

Retrieval is a scatter-gather approach, invoking several components. The first is a lightweight scoring and targeting filter. Scoring estimates how valuable the content is using very simple models. Targeting restricts the ads to certain subsets of users, based on criteria selected by advertisers: for example, targeting ads based on the user's location.

The next steps are around budgeting and pacing. If an ad has finished all its budget, it should not be retrieved. Pacing is a related concept: it is a means of spreading the ad spend across time. For example, if an ad has a $100 budget, the advertiser doesn't want to spend this $100 in the first hour, because that might not result in the best value. Advertising platforms tend to match pacing to daily patterns of traffic on their platform.

To ensure that there's a diversity of ads, deduping limits the number of ads that an advertiser can contribute: the platform shouldn't overwhelm a feed with ads only from a single advertiser. For example, only the top-K candidates per advertiser are allowed to be passed into the next stage. Finally, since this is a scatter-gather approach, there may be different retrieval sources whose results must be blended together before sending to further down in the funnel.

The next step is candidate selection and recent advancements in this field. Traditionally, candidate retrievers could be as simple as

Figure 4: Standard Query during Retrieval - Scatter Gather

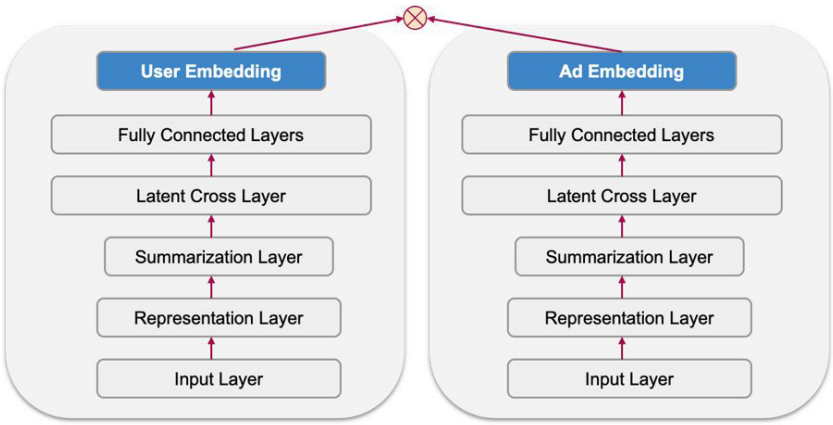


Figure 5: Two-Tower DNN [P Covington et. al, 2016]

matching keywords or ad copy text. As the system grows more complex, this becomes harder to maintain and makes it harder to iterate.

In 2016, YouTube had a seminal paper which changed the way these retrieval systems worked, by introducing Two-Tower Deep Neural Networks. The idea is to learn latent representations of users and content based on their features. These representations and features are kept separate from each other in the model. In the end, however, if the user has engaged with a content item, those representations should be very close to each other, and so that's the training objective of the model.

The benefit of this model is that ad embeddings can be precomputed, cached, and indexed offline. The ads database builds an index by passing each ad through the ad "tower" of the model to generate its embedding. Once the ad is indexed during the serving time, the retrieval server needs to only call the user part of the model, then utilize approximate nearest neighbor search algorithms like HNSW, to find relevant ads in the ad database index.

The Ranking Model

Next up is the ranking model. Beginning in 2014, the models were simplistic ones like logistic regression. The next step that happened in this evolution, to make the models more expressive, Pinterest moved from simplistic solutions to more complex models such as GBDT plus logistic regression solutions.

There are four types of features that a model can have: user features; content features; interaction between the user and the content in the history; and finally, events happening during this impression time. Models should learn some nonlinear interactions between these features, and GBDTs are good at it. Also, the model retains a logistic regression framework, which is a linear model which captures high cardinality features. Note that GBDTs aren't good with such kinds of features.

Very soon, Pinterest had around 60 models in production. These models were growing and the product was growing. It became complex to maintain all these models, leading to long cycles

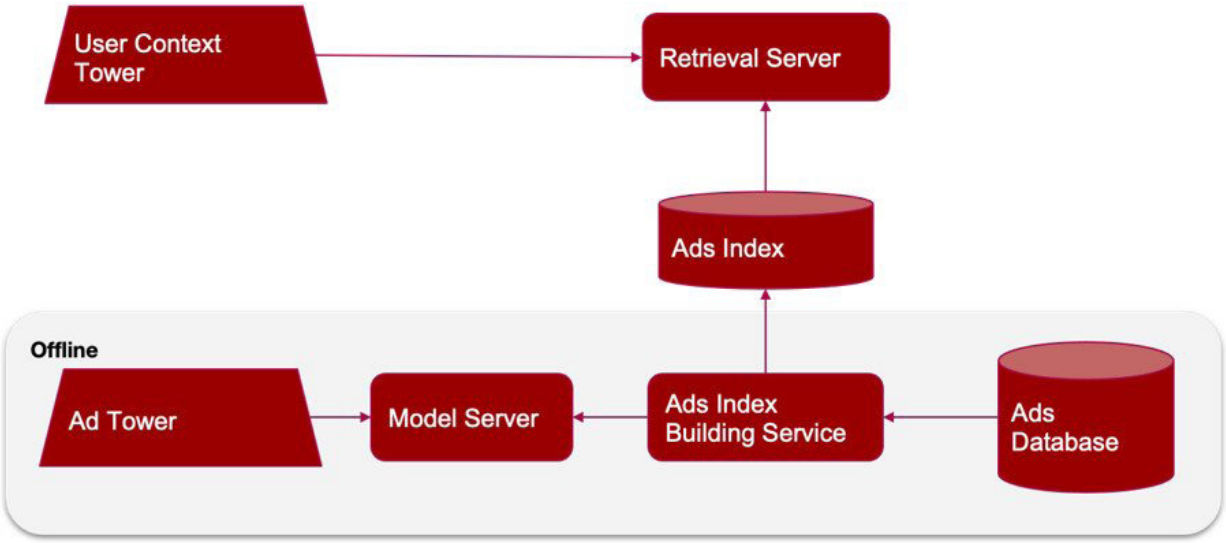


Figure 6: Two Tower Model Deployment

in feature adoption and deletion, which lead to suboptimal systems.

Also, around this time, machine learning systems did not easily support model serving. Pinterest was using a different language or framework for training models vs. serving them. For example, Pinterest used to train using XGBoost, then translate that into a TensorFlow model, then translate that into C++, Pinterest's serving language. These kinds of hops in the system lead to suboptimality and longer cycles to develop new features.

Finally, new ad groups were constantly being created or deleted: ads might have maybe a one or two month time window when they're alive. Pinterest needed the models to be responsive, so that they could be trained more incrementally on new data distributions that are coming in. However, the GBDT models are static: there is no way to train them incrementally. Deep neural networks (DNN), on the other hand, have incremental power to be trained.

The next iteration was replacing the GBDT with DNN approaches. A DNN brings many benefits, but they are more complex models. One of the things that changed here is that previous traditional machine learning algorithms relied more on feature engineering by hand, where engineers would define what two features might be related. The models inherently couldn›t learn feature interactions by themselves. In DNNs architectures, the model can learn these interactions.

Most recommendation models in the industry share a similar multi-layer architecture. The first is the representation layer, which is where platforms define features and how the model can understand those features. In this particular scenario, for DNNs, feature processing is very important. If the feature scale is different across different features, the model can break, so the layer includes logic for squashing or clipping the values, or doing some normalization to the features.

Next, if two features are related to each other, the model can summarize them together and learn a common embedding. After that are multiplicative cross-layers, which learn feature interactions, followed by fully-connected layers.

Another benefit of DNNs is multitask learning across different objectives. Network weights are

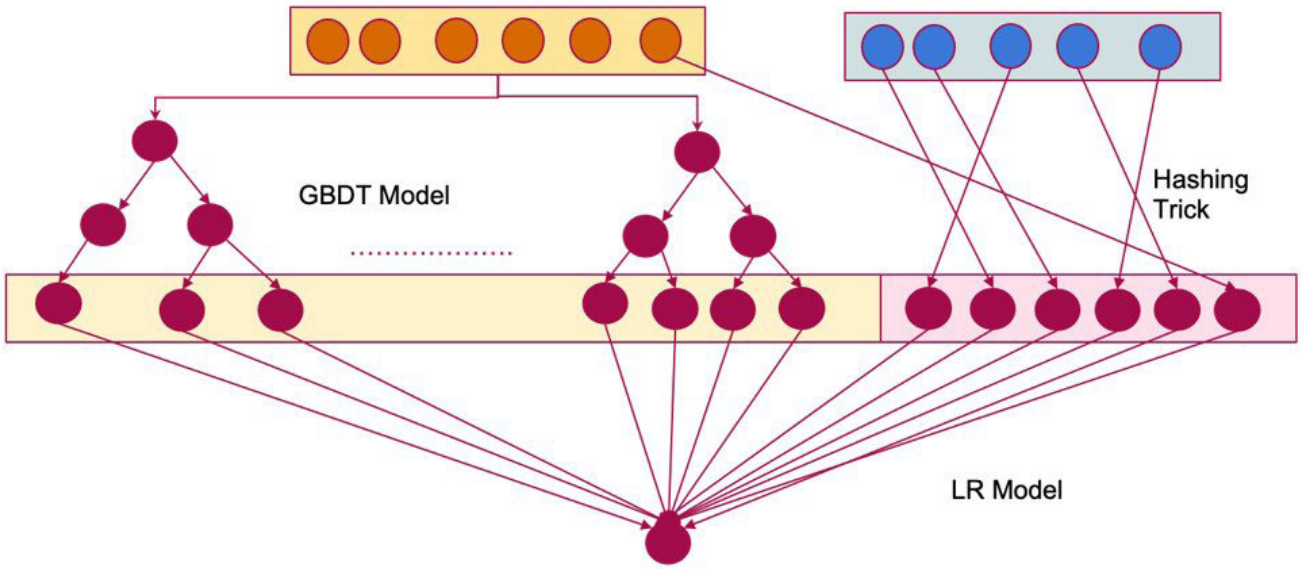


Figure 7: GBDT + Logistic Regression Ensemble Models

shared across different objectives like clicks, repins, or whatever other metrics there may be on the platform, removing the need to train different models for different objectives.

The next iteration of the model utilizes the sequence of activities that the user is doing on the platform. Let's assume that a user could have interacted with multiple pins on the platform or multiple images on the platform, which could be food related pins, home decor, or travel related pins. The idea is, can the platform use this representation of what the user is doing to define what the user might do next?

To implement this, Pinterest turned to the Transformer DNN architecture. Transformers can encode very powerful information about feature interactions. A key parameter of the model is the maximum sequence length. As sequence length increases, model size grows quadratically, which impacts serving capacities.

Increasing the sequence length to, say, 100 events, makes the complex features discussed above impractical. Instead, the model uses simple features like: what's that action? Did the user click or not? Very simple features, but a longer sequence enables the model to have better capacities.

The latest model architecture for offline user representation at Pinterest is based on a Transformer encoder called PinnerFormer. This component takes inputs from user engagements in the past: say from yesterday to a year back. All of these engagements are encoded in an offline manner to learn an embedding for each user, which can then be used as a feature input into a downstream DNN model.

Another input to that model is the real-time sequence coming from current user engagements. A combination of these two can learn what the user is doing on the platform. Utilizing these sequences, which is taking inspiration from the NLP domain, is what's powering Pinterest's recommendation system.

MLOps at Pinterest

In the recommendation system overall, and how it's deployed and operated in production, machine learning is a very small piece of it. There are many other things to consider, such as: how to make sure that dev teams can iterate faster? How to make sure the serving infrastructure can support

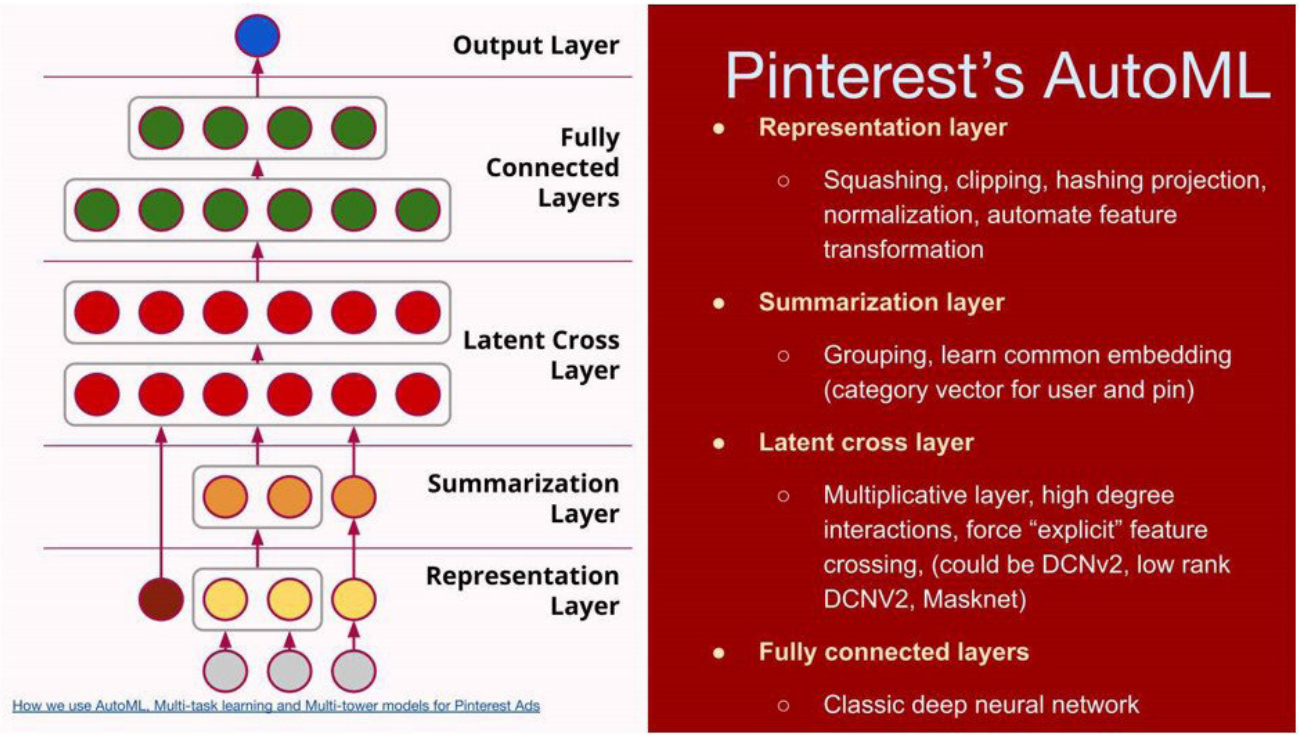


Figure 8: Pinterest's AutoML Architecture

the models? How to manage resources? How to store and verify the data? Having all those kinds of checks are very important.

In the past, every team at Pinterest used to have many different pipelines: everyone was rearchitecting the same wheel. Pinterest needed to do this in a more scalable manner. That's where most of the iterations happened in the last year. Pinterest built a unified, Pytorch-based ML framework (MLEnv) that provides Docker images and traditional CI/CD services. The part where the user writes the code is very small, and integration between various MLOps components is done seamlessly through API based solutions, which allows teams to iterate faster.

The standard model deployment process uses MLflow, which is an open source solution. As these models are moved into the production pipeline, they are versioned, so that teams can do rollbacks easily. Also, models are reproducible: MLflow has a UI where users can see what parameters went into training. If a team needs to retrain and reevaluate the training process, that's easy to do.

Testing and Monitoring

The first testing step is integration testing. When a code change is written, Pinterest can test it in the production environment through shadow traffic to see what would happen if the change were deployed. Automatically capturing metrics ensures that nothing is missed during the testing process. There is also a debugging system that can replay how a particular request would look based on the serving of a particular model version.

The next step is around how the code is released once it's merged into the system. Pinterest follows the standard process of canary, staging, and production pipelines. Each of these stages monitors real-time metrics that the business cares about. If there is a deviation day-over-day, or there's a devi-

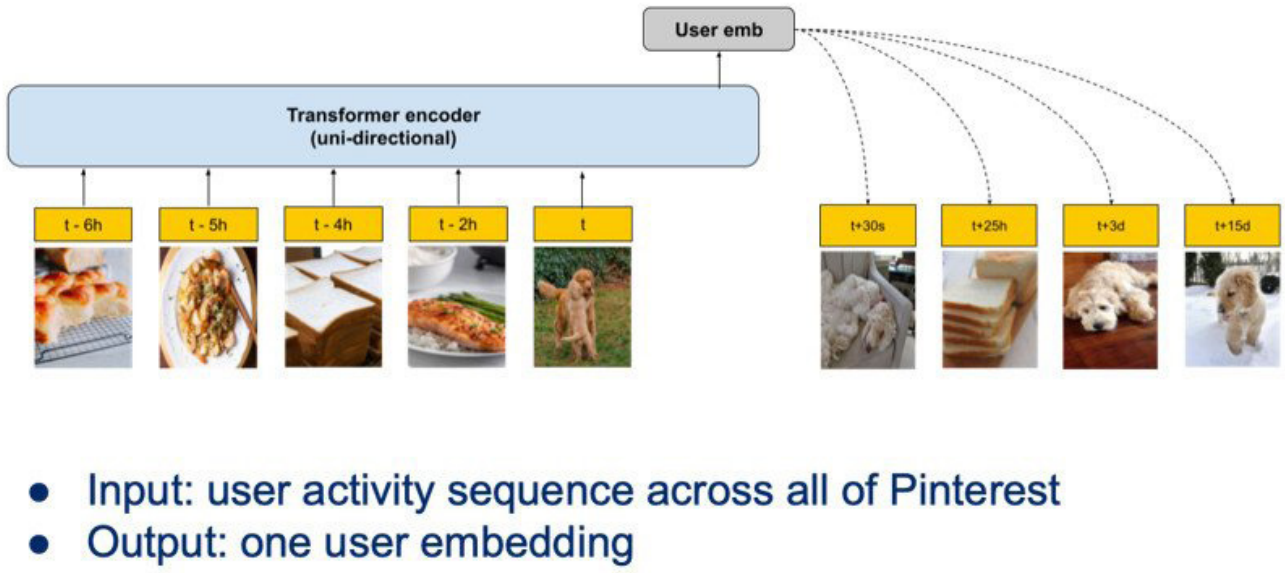


Figure 9: PinnerFormer: Sequence Modeling for User Representation at Pinterest

ation between production and another environment, the deployment would be stopped and rolled back in a seamless manner.

Finally, despite all these safeguard, bugs could still escape. Also, advertisers might have different behaviors. So Pinterest has real-time monitoring which is capturing the day-over-day and week-over-week patterns into the system along different dimensions, which could be revenue, insertion rates, and QPS.

ML Workflow Validation and Monitoring

Besides monitoring the production metrics, ML workflows have additional monitoring requirements. The first step is looking at the training datasets that are being fed into the model, and defining coverage and alerting on

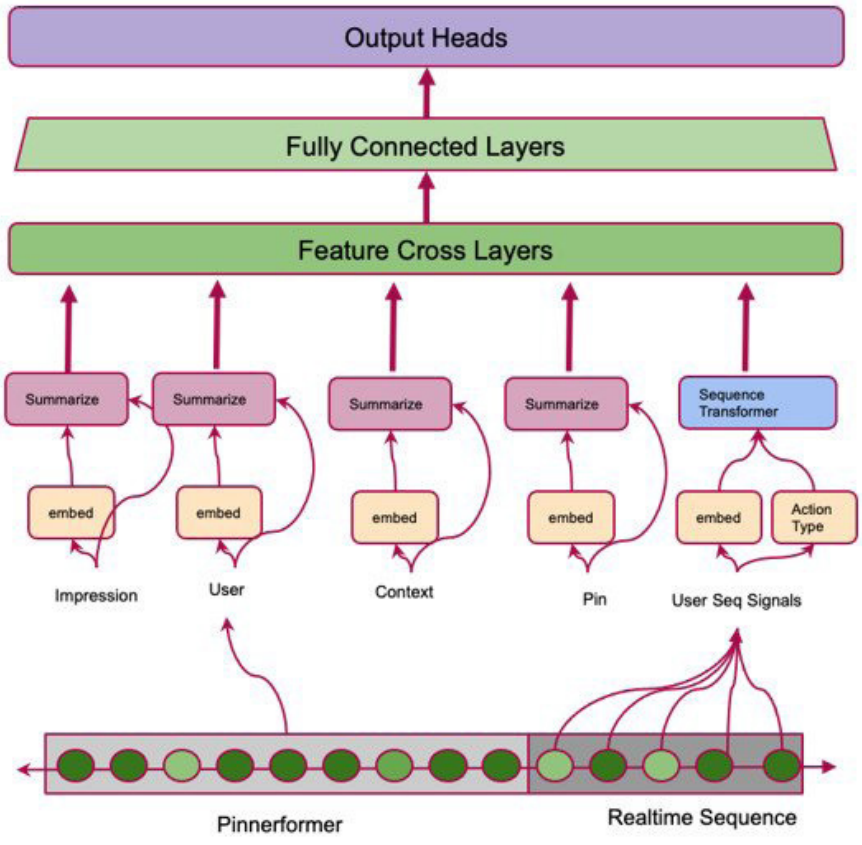


Figure 10: Combining Long Sequences

top of that. For example, monitoring features and how they change over time, and ensuring that features are fresh.

The next set of testing that happens is around the offline model evaluation. Once there is a trained model, developers need to check whether this model will make the right predictions. Pinterest captures model metrics like AUC, but they also capture predictions, to see if there are spikes into the predictions. If there are, these can halt the model validation processes. They also monitor for prediction spikes in production.

To be able to debug the system, Pinterest has developed several different tools. One key is to have visibility into the ad delivery funnel: retrieval, budgeting, indexing, and advertiser. Pinterest's tools help them locate where the ad is being removed from the funnel.

For example, suppose an ad is not being shown very often. If it's not shown on the serving side, it might be that the ad is very low quality, or this ad is not competitive in the auction. Another scenario might be an advertiser only wants to show the ad to particular users; this is a very tight retrieval scenario, so that's why the ad might not be showing.

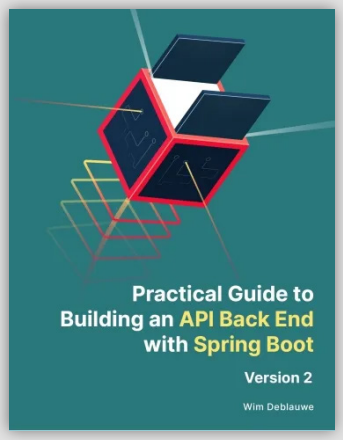
Serving Large Models

Another goal is to make sure that the serving infrastructure has low latency, which enables Pinterest to score more ads. One way to improve latency is to move to GPU serving if the models are more complex. If that's not an option, there are optimization techniques, such as quantizing models or knowledge distillation, to improve latency, usually at the cost of inference accuracy.

Conclusion

Mudgal presented an overview of Pinterest's ad serving system and how they use ML at scale in production. He also discussed how Pinterest monitors and tests their models before and after they are deployed to production. Mudgal provided several insights that the audience can apply to their systems to overcome similar challenges.

Curious about previous issues?



If you are eager to learn about Spring Boot, REST APIs, security, validation, and testing, consider giving this updated book a try. You are guaranteed to learn something new!

Our reports have delved deep into the latest advancements, challenges, and future directions. We have aimed to provide insight into topics such as hybrid working, monoliths vs microservices vs moduliths, platform engineering, large language models (LLMs), and the state-of-the-art within the Java ecosystem.

In the InfoQ “Data Engineering Innovations” eMag, you’ll find up-to-date case studies and real-world data engineering solutions from technology SME’s and leading data practitioners in the industry.

InfoQ