The Scaling Journey of LinkedIn

BYTEBYTEGO
MAY 28, 2024

10 Insights On Real-World Container Usage (Sponsored)

As organizations have scaled their containerized environments, many are now exploring the next technology frontier of containers by building next-gen applications, enhancing developer productivity, and optimizing costs.

Datadog analyzed telemetry data from over 2.4 billion containers to understand the present container landscape, with key insights into:

- Trends in adoption for technologies such as serverless containers and GPU-based compute on containers
- How organizations are managing container costs and resources through ARM-based compute and horizontal pod autoscaling
- Popular workload categories and languages for containers

LinkedIn is one of the biggest social networks in the world with almost a billion members.

But the platform had humble beginnings.

The idea of LinkedIn was conceived in Reid Hoffman's living room in 2002 and it was officially launched in May 2003. There were 11 other co-founders from Paypal and Socialnet who collaborated closely with Reid Hoffman on this project.

The initial start was slow and after the first month of operation, LinkedIn had just around 4300 members. Most of them came through personal invitations by the founding members.

However, LinkedIn's user base grew exponentially over time and so did the content hosted on the platform. In a few years, LinkedIn was serving tens of thousands of web pages every second of every day to users all over the world.

This unprecedented growth had one major implication.

LinkedIn had to take on some extraordinary challenges to scale its application to meet the growing demand. While it would've been tough for the developers involved in the multiple projects, it's a gold mine of lessons for any developer.

In this article, we will look at the various tools and techniques LinkedIn adopted to scale the platform.

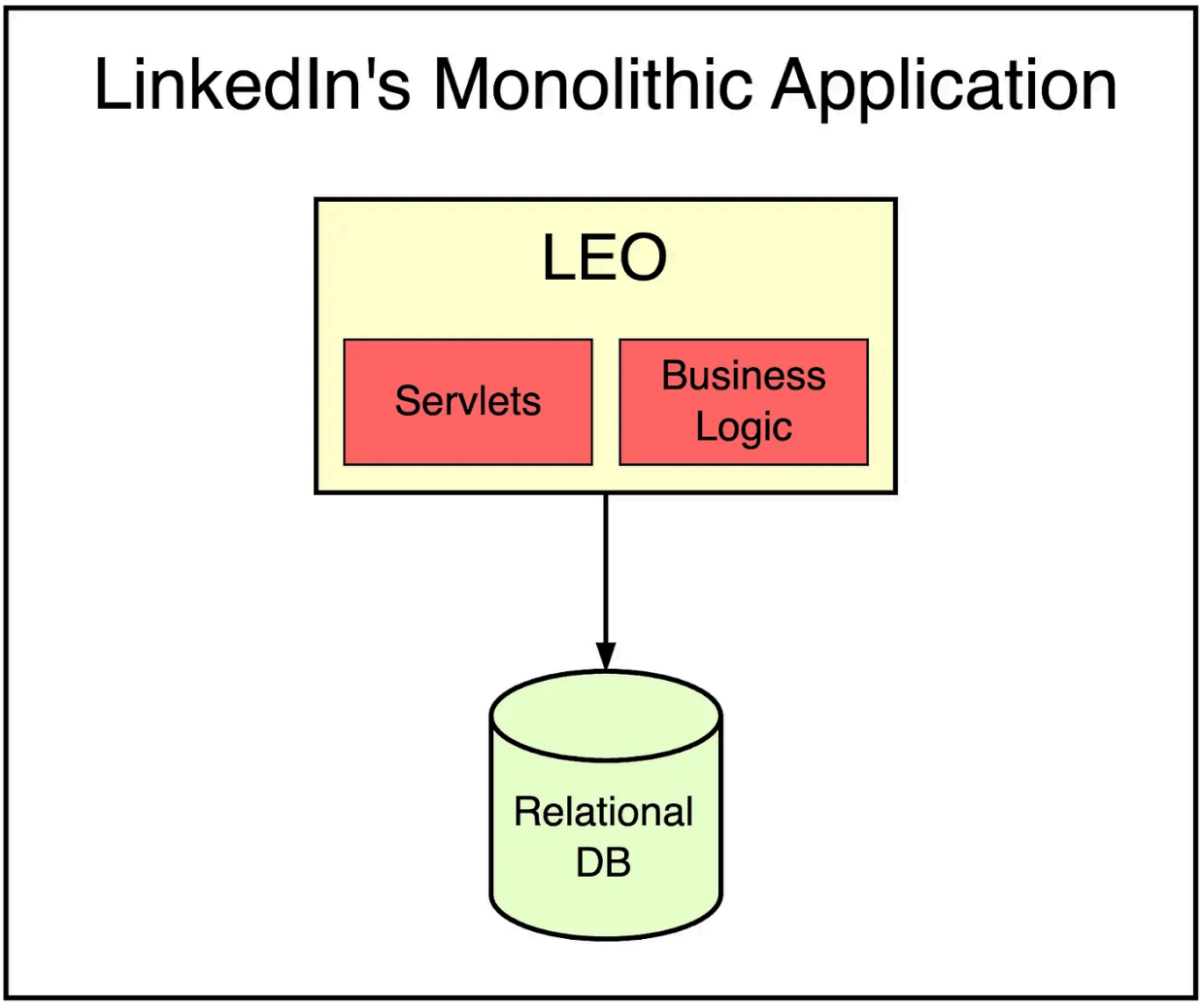
Humble Beginning with Leo

Like many startups, LinkedIn also began life with a **monolithic architecture**.

There was one big application that took care of all the functionality needed for the website. It hosted web servlets for the various pages, handled business logic, and also connected to the database layer.

This monolithic application was internally known as Leo. Yes, it was as magnificent as MGM's Leo the Lion.

The below diagram represents the concept of Leo on a high level.



However, as the platform grew in terms of functionality and complexity, the monolith wasn't enough.

The First Need of Scaling

The first pinch point came in the form of two important requirements:

- Managing member-to-member connections
- Search capabilities

Let's look at both.

Member Graph Service

A social network depends on connections between people.

Therefore, it was critical for LinkedIn to effectively manage member to member connections. For example, LinkedIn shows the graph distance and common connections whenever we view a user profile on the site.

To display this small piece of data, they needed to perform low-latency graph computations, creating the need for a system that can query connection data using **in-memory graph traversals**. The in-memory requirement is key to realizing the performance goals of the system.

Such a system had a completely different usage profile and scaling need as compared to Leo.

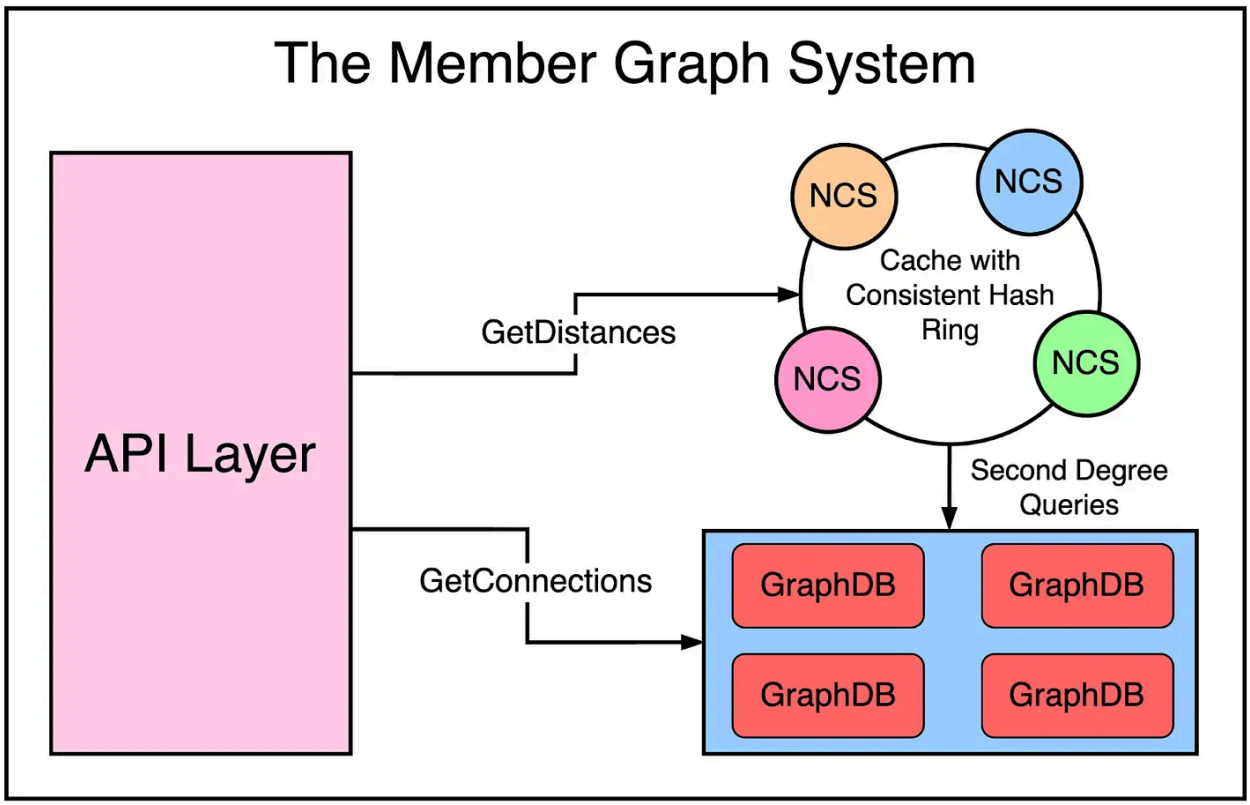
Therefore, the engineers at LinkedIn built a **distributed and partitioned graph system** that can store millions of members and their connections. It could also handle hundreds of thousands of queries per second (QPS).

The system was called Cloud and it happened to be the first service at LinkedIn. It consisted of three major subcomponents:

1. **GraphDB** - a partitioned graph database that was also replicated for high availability and durability
2. **Network Cache Service** - a distributed cache that stores a member's network and serves queries requiring second-degree knowledge

3. **API Layer** - the access point for the front end to query the data.

The below diagram shows the high-level architecture of the member graph service.



To keep it separate from Leo, LinkedIn utilized Java RPC for communication between the monolith and the graph service.

Search Service

Around the same time, LinkedIn needed to support another critical functionality - **the capability to search people and topics**.

It is a core feature for LinkedIn where members can use the platform to search for people, jobs, companies, and other professional resources. Also, this search feature should aim to provide deeply personalized search results based on a member's identity and relationships.

To support this requirement, a new search service was built using **Lucene**.

Lucene is an open-source library that provides three functionalities:

- Building a search index

- Searching the index for matching entities
- Determining the importance of these entities through relevant scores

Once the search service was built, both the monolith and the new member graph service started feeding data into this service.

While the building of these services solved key requirements, the continued growth in traffic on the main website meant that Leo also had to scale.

Let's look at how that was achieved.

Scaling Leo

As LinkedIn grew in popularity, the website grew and Leo's roles and responsibilities also increased. Naturally, the once-simple web application became more complex.

So - how was Leo scaled?

One straightforward method was to spin up multiple instances of Leo and run them behind a Load Balancer that routes traffic to these instances.

It was a nice solution but it only involved the application layer of Leo and not the database. However, the increased workload was negatively impacting the performance of LinkedIn's most critical system - its **member profile database** that stored the personal information of every registered user. Needless to say, this was the heart of LinkedIn.

A quick and easy fix for this was going for classic vertical scaling by throwing additional compute capacity and memory for running the database. It's a good approach to buy some time and get some breathing space for the team to think about a long-term solution to scaling the database.

The member profile database had one major issue. It handled both read and write traffic, resulting in a heavy load.

To scale it out, the team turned to **database replication**.

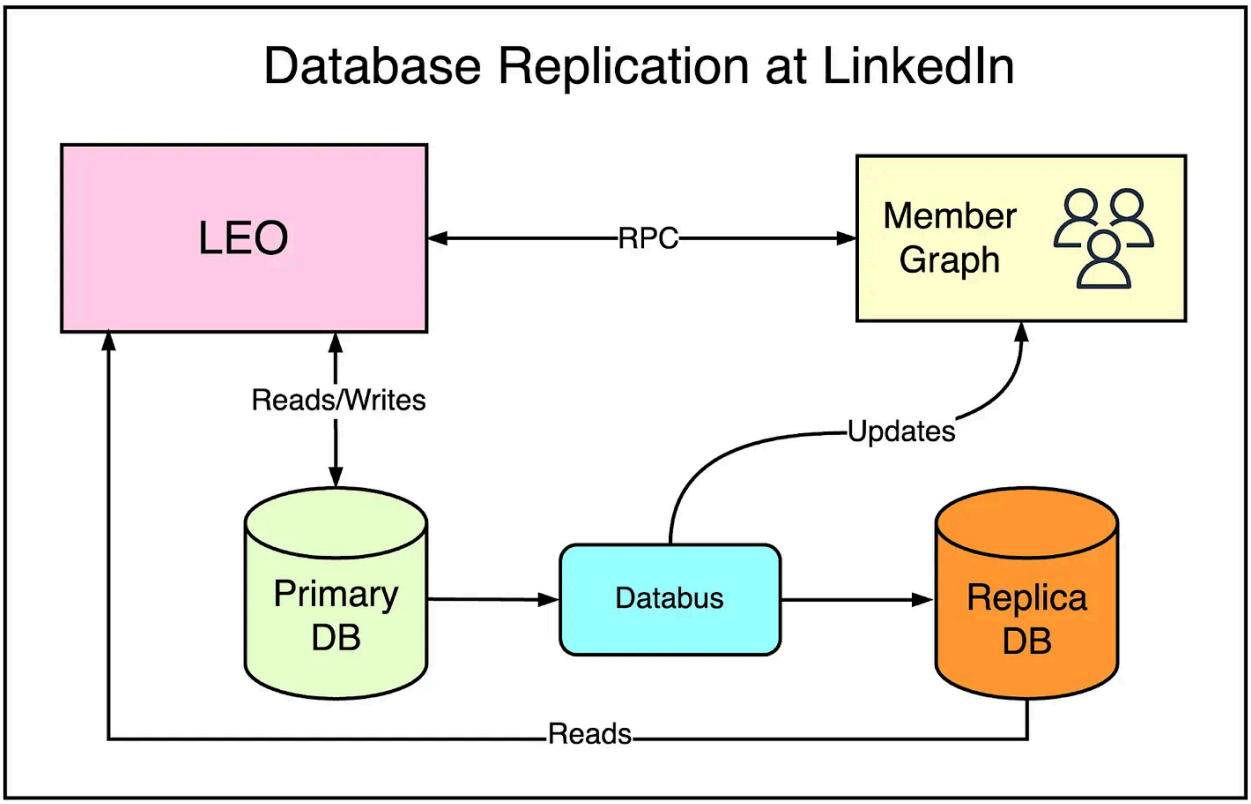
New replica databases were created. These replicas were a copy of the primary database and stayed in sync with the primary using Databus. While writes were still

handled by the primary database, the trick was to send the majority of read requests to the replica databases.

However, data replication always results in some amount of replication lag. If a request reads from the primary database and the replica database at the same time, it can get different results because the replication may not have completed. A classic example is a user updating her profile information and not able to see the updated data on accessing the profile just after the update.

To deal with issues like this, special logic was built to decide when it was safe or consistent to read from the replica database versus the primary database.

The below diagram tries to represent the architecture of LinkedIn along with database replication



While replication solved a major scaling challenge for LinkedIn, the website began to see more and more traffic. Also, from a product point of view, LinkedIn was evolving rapidly.

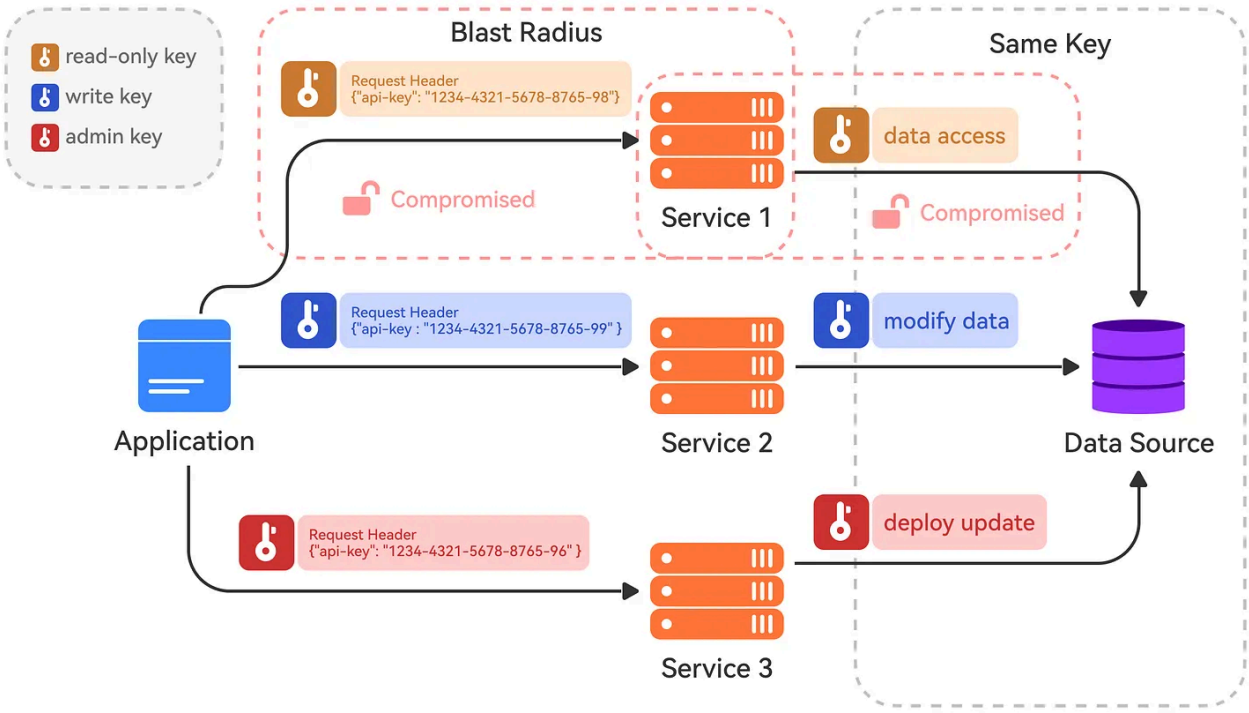
It created two major challenges:

- Leo was often going down in production and it was becoming more difficult to recover from failures
- Releasing new features became tough due to the complexity of the monolithic application

High availability is a critical requirement for LinkedIn. A social network being down can create serious ripple effects for user adoption. It soon became obvious that they had to kill Leo and break apart the monolithic application into more manageable pieces.

Latest articles

If you're not a paid subscriber, here's what you missed.



1. API Security Best Practices
2. A Crash Course in GraphQL
3. HTTP1 vs HTTP2 vs HTTP3 - A Deep Dive
4. Unlocking the Power of SQL Queries for Improved Performance
5. What Happens When a SQL is Executed?

To receive all the full articles and support ByteByteGo, consider subscribing:

Killing Leo with Service-Oriented Architecture

While it sounds easy to break apart the monolithic application, it's not easy to achieve in practice.

You want to perform the migration in a seamless manner without impacting the existing functionality. Think of it as changing a car's tires while it is moving on the highway at 60 miles per hour.

The engineers at LinkedIn started to extract functionalities from the monolith in their own separate services. Each service contained APIs and business logic specific to a particular functionality.

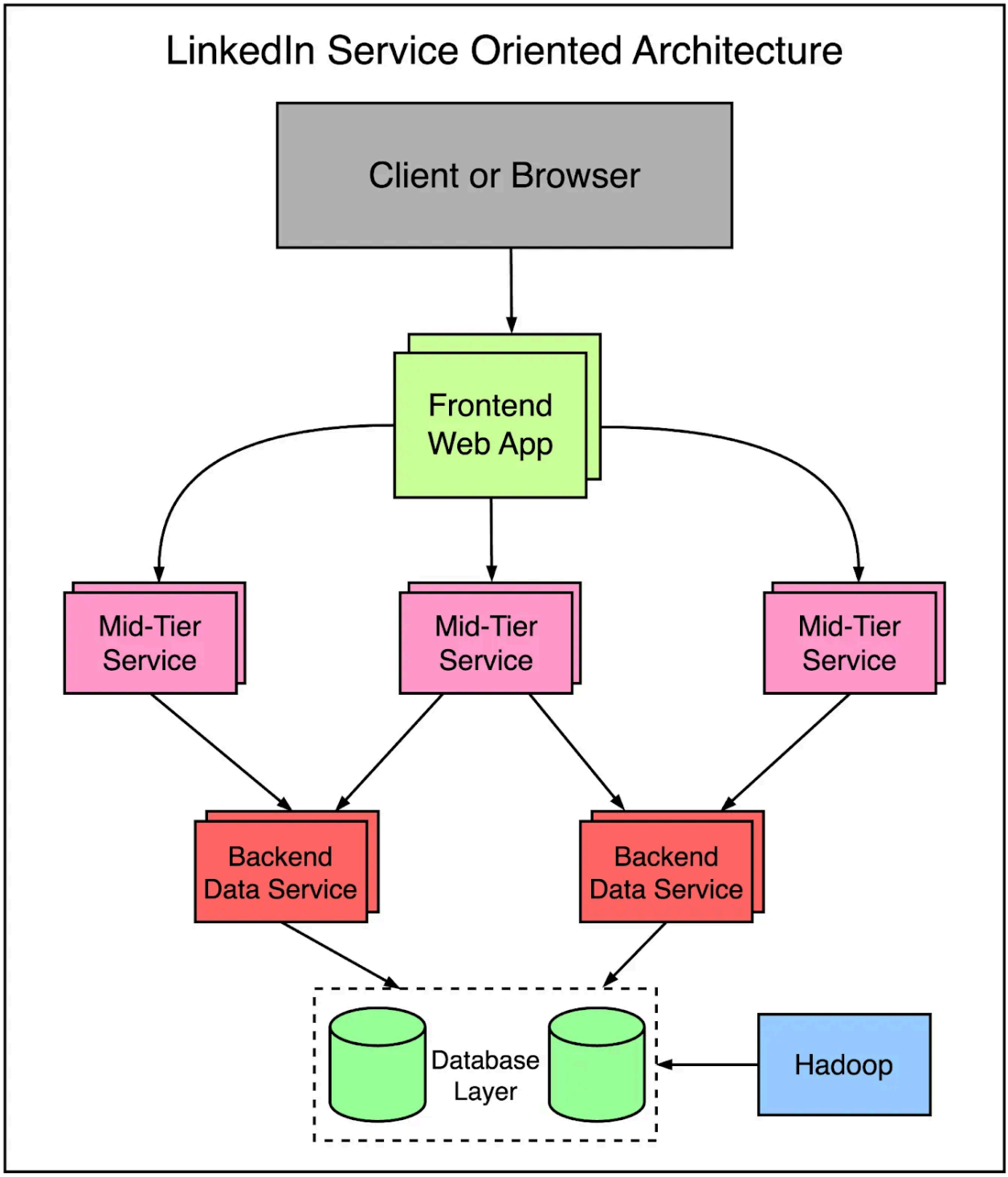
Next, services to handle the **presentation layer** were built such as public profiles or recruiter products. For any new product, brand-new services were created completely outside of Leo.

Over time, the effort towards SOA led to the emergence of vertical slices where each slice handled a specific functional area.

- The frontend servers fetched data from different domains and handled the presentation logic to build the HTML via JSPs.
- The mid-tier services provided API access to data models.
- The backend data services provided consistent access to the database.

By 2010, LinkedIn had already built over 150 separate services and by 2015, they had over 750 services.

The below diagram represents a glimpse of the SOA-based design at LinkedIn:



At this point, you may wonder what was the benefit of this massive change.

- First, these services were built in a **stateless manner**. Scaling can be achieved by spinning up new instances of a service and putting them behind a load balancer. Such an approach is known as horizontal scaling and it was more cost-effective compared to scaling the monolithic application.
- Second, each service was expected to define how much load it could take and the engineering team was able to build out early provisioning and performance

monitoring capabilities to support any deviations.

Managing Hypergrowth with Caching

It's always a good thing for a business owner to achieve an exponential amount of growth.

Of course, it does create a bunch of problems. Happy problems but still problems that must be solved.

Despite moving to service-oriented architecture and going for replicated databases, LinkedIn had to scale even further.

This led to the adoption of caching.

Many applications started to introduce mid-tier caching layers like memcached or couchbase. These caches were storing derived data from multiple domains. Also, they added caches to the data layers by using Voldemort to store precomputed results when appropriate.

However, if you've worked with caching, you would know that caching brings along with it a bunch of new challenges in terms of invalidations, managing consistency, and performance.

Over time, the LinkedIn team got rid of many of the mid-tier caches.

Caches were kept close to the data store in order to reduce the latency and support horizontal scalability without the cognitive load of maintaining multiple caching layers.

Data Collection with Kafka

As LinkedIn's footprint grew, it also found itself managing a huge amount of data.

Naturally, when any company acquires a lot of data, it wants to put that data to good use for growing the business and offering more valuable services to the users. However, to make meaningful conclusions from the data, they have to collect the data and bring it in one place such as a data warehouse.

LinkedIn started developing many custom data pipelines for streaming and queuing data from one system to another.

Some of the applications were as follows:

- Aggregating logs from every service
- Collecting data regarding tracking events such as pageviews
- Queuing of emails for LinkedIn's inMail messaging system
- Keeping the search system up to date whenever someone updates their profile

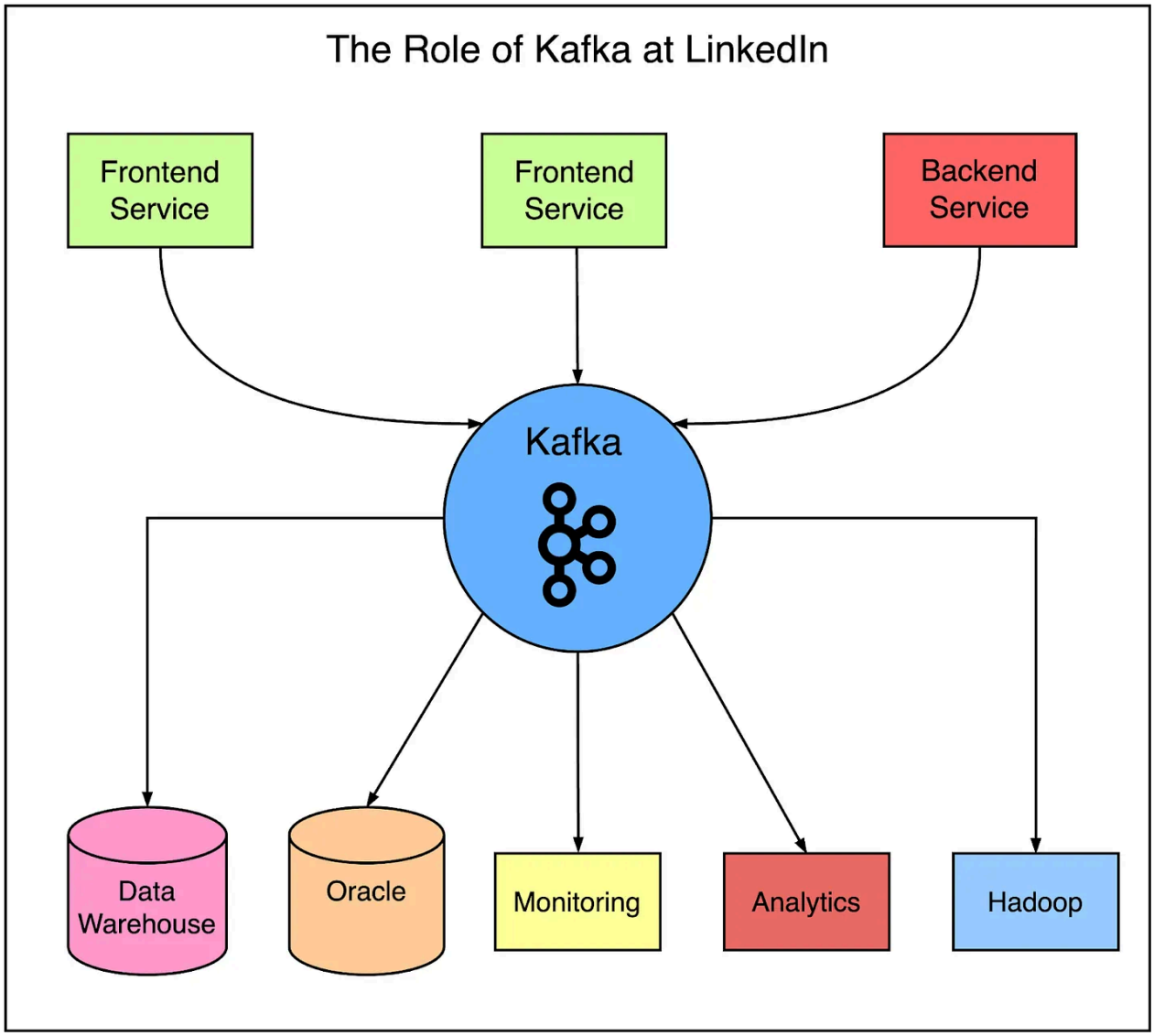
As LinkedIn grew, it needed more of these custom pipelines and each individual pipeline also had to scale to keep up with the load.

Something had to be done to support this requirement.

This led to the development of Kafka, a distributed pub-sub messaging platform. It was built around the concept of a commit log and its main goal was to enable speed and scalability.

Kafka became a universal data pipeline at LinkedIn and enabled near real-time access to any data source. It empowered the various Hadoop jobs and allowed LinkedIn to build real-time analytics, and improve site monitoring and alerting.

See the below diagram that shows the role of Kafka at LinkedIn.



Over time, Kafka became an integral part of LinkedIn's architecture. Some latest facts about Kafka adoption at LinkedIn are as follows:

- Over 100 Kafka clusters with more than 4000 brokers
- 100K topics and 7 million partitions
- 7 trillion messages handled per day

Scaling the Organization with Inversion

While scaling is often thought of as a software concern, LinkedIn realized very soon that this is not true.

At some time, you also need to scale up at an organizational level.

At LinkedIn, the organizational scaling was carried out via an internal initiative called **Inversion**.

Inversion put a pause on feature development and allowed the entire engineering organization to focus on improving the tooling and deployment, infrastructure and developer productivity. In other words, they decided to focus on improving the developer experience.

The goal of Inversion was to increase the engineering capability of the development teams so that new scalable products for the future could be built efficiently and in a cost-effective way.

Let's look at a few significant tools that were built as part of this initiative:

Rest.li

During the transformation from Leo to a service-oriented architecture, the extracted APIs were based on Java-based RPC.

Java-based RPC made sense in the early days but it was no longer sufficient as LinkedIn's systems evolved into a polyglot ecosystem with services being written in Java, Node.js, Python, Ruby and so on. For example, it was becoming hard for mobile services written in Node.js to communicate with Java object-based RPC services.

Also, the earlier APIs were tightly coupled with the presentation layer, making it difficult to make changes.

To deal with this, the LinkedIn engineers created a new API model called **Rest.li**.

What made Rest.li so special?

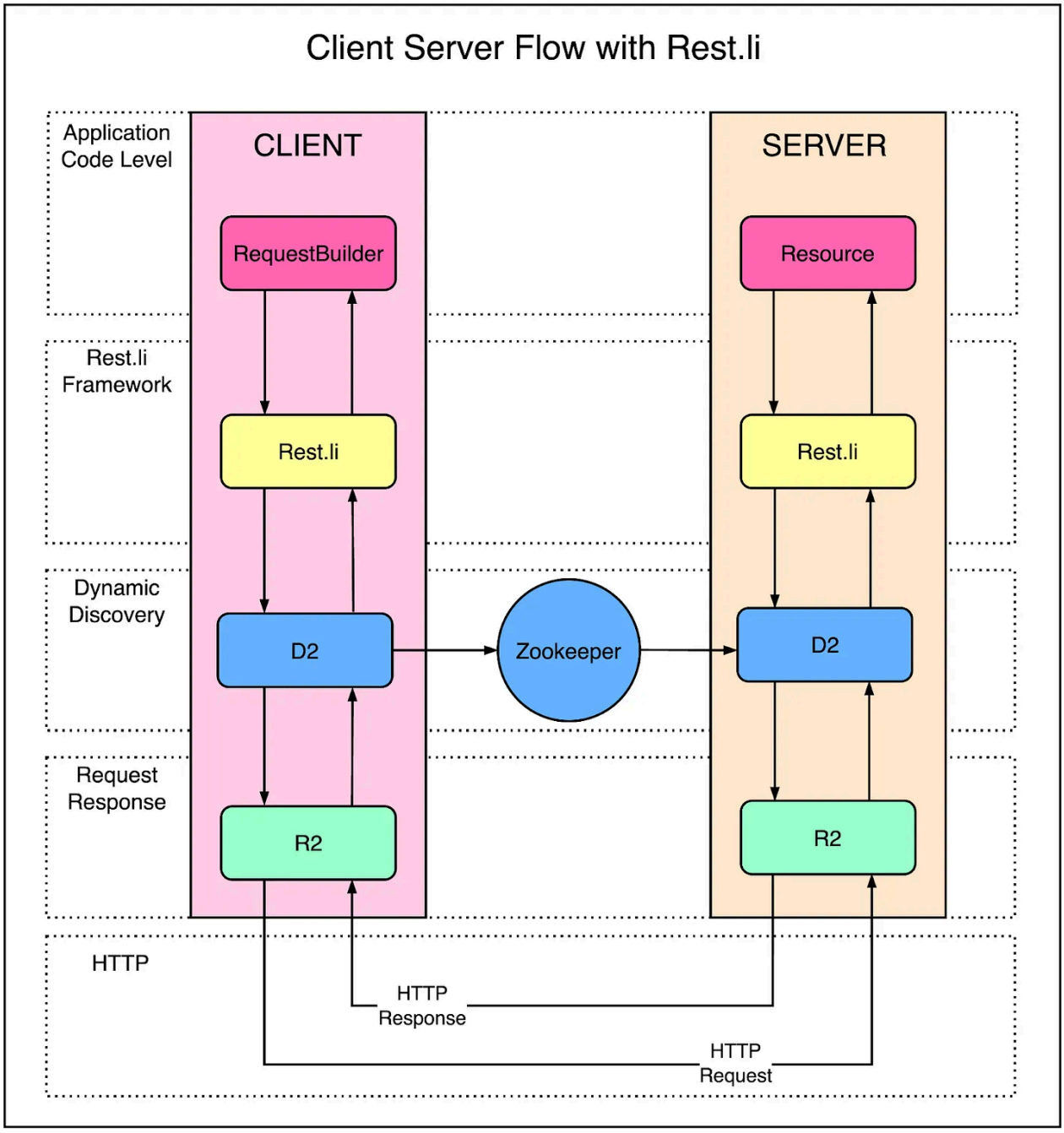
Rest.li was a framework for developing RESTful APIs at scale. It used simple JSON over HTTP, making it easy for non-Java-based clients to communicate with Java-based APIs.

Also, Rest.li was a step towards a data-model-based architecture that brought a consistent API model across the organization.

To make things even more easy for developers, they started using Dynamic Discovery (D2) with Rest.li services. With D2, there was no need to configure URLs for each

service that you need to talk to. It provides multiple features such as client-based load balancing, service discovery and scalability.

The below diagram shows the use of Rest.li along with Dynamic Discovery.



Super Blocks

A service-oriented architecture is great for decoupling domains and scale out services independently.

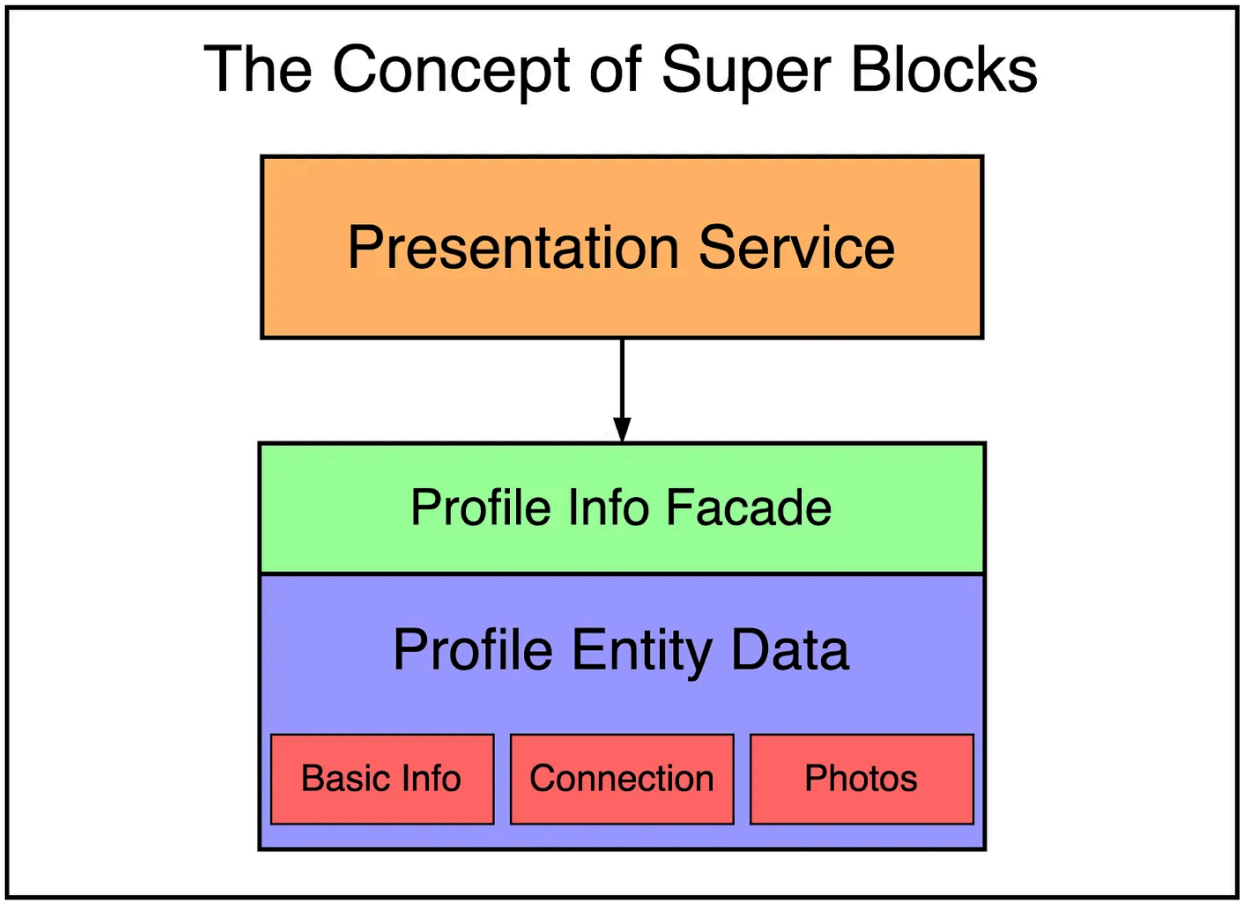
However, there are also downsides.

Many of the applications at LinkedIn depend on data from multiple sources. For example, any request for a user's profile page not only fetches the profile data but includes other details such as photos, connections, groups, subscription information, following info, long-form blog posts and so on.

In a service-oriented architecture, it means making hundreds of calls to fetch all the needed data.

This is typically known as the "call graph" and you can see that this call graph can become difficult to manage as more and more services are created.

To mitigate this issue, LinkedIn introduced the concept of a **super block**.



A super block is a grouping of related backend services with a single access API.

This allows teams to create optimized interfaces for a bunch of services and keep the call graph in check. You can think of the super block as the implementation of the facade pattern.

Multi-Data Center

In a few years after launch, LinkedIn became a global company with users joining from all over the world.

They had to scale beyond serving traffic from just one data center. Multiple data centers are incredibly important to maintain high availability and avoid any single point of failure. Moreover, this wasn't needed just for a single service but the entire website.

The first move was to start serving public profiles out of two data centers (Los Angeles and Chicago).

Once it was proven that things work, they enhanced all other services to support the below features:

- Data replication
- Callbacks from different origins
- One-way data replication events
- Pinning users to geographically-close data centers

As LinkedIn has continued to grow, they have migrated the edge infrastructure to Azure Front Door (AFD). For those who don't know, AFD is Microsoft's global application and content delivery network and migrating to it provided some great benefits in terms of latency and resilience.

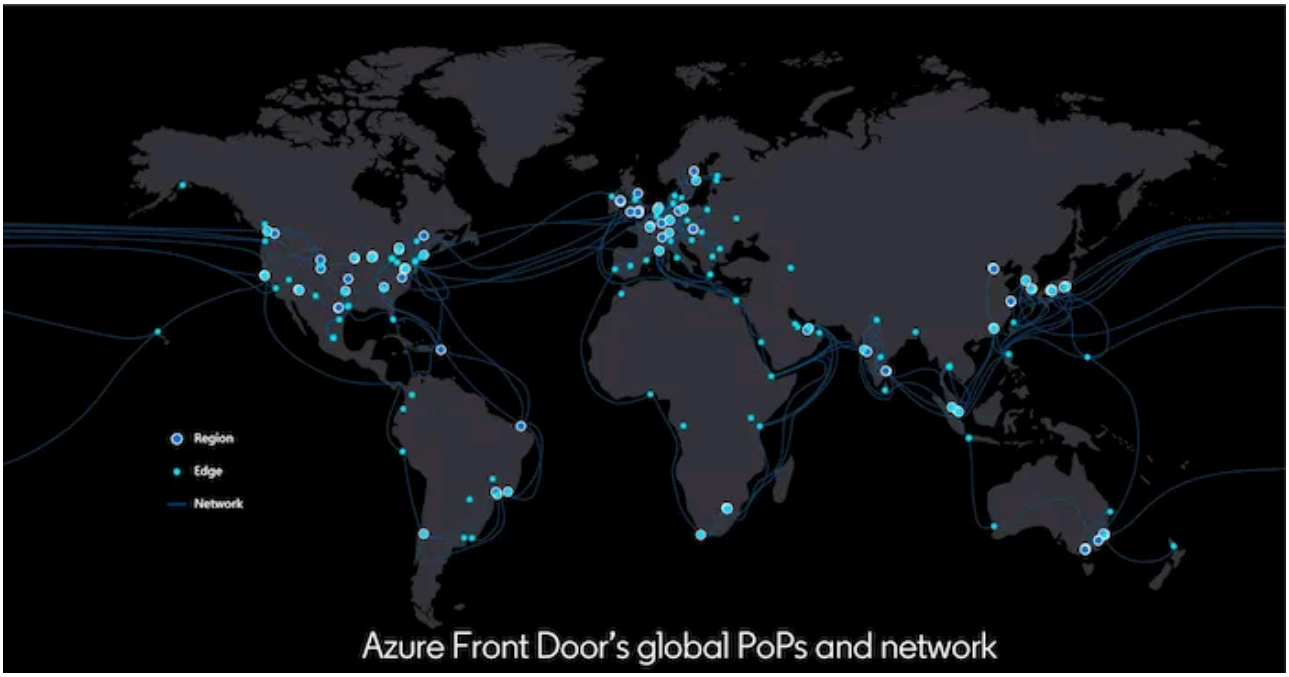


Image Source: Scaling LinkedIn's Edge with Azure Front Door

This move scaled them up to 165+ Points of Presence (PoPs) and helped improve median page load times by up to 25 percent.

The edge infrastructure is basically how our devices connect to LinkedIn today. Data from our device traverses the internet to the closest PoP that houses HTTP proxies that forward those requests to an application server in one of the LinkedIn data centers.

Advanced Developments Around Scalability

Running an application as complex and evolving as LinkedIn requires the engineering team to keep investing into building scalable solutions.

In this section, we will look at some of the more recent developments LinkedIn has undergone.

Real Time Analytics with Pinot

A few years ago, the LinkedIn engineering team hit a wall with regards to analytics

The scale of data at LinkedIn was growing far beyond what they could analyze. The analytics functionality was built using generic storage systems like Oracle and

Voldemort. However, these systems were not specialized for OLAP needs and the data volume at LinkedIn was growing in both breadth and depth.

At this point, you might be wondering about the need for real-time analytics at LinkedIn.

Here are three very important use-cases:

1. The **Who's Viewed Your Profile** is LinkedIn's flagship analytics product that allows members to see who has viewed their profile in real-time. To provide this data, the product needs to run complex queries on large volumes of profile view data to identify interesting insights.
2. **Company Page Analytics** is another premium product offered by LinkedIn. The data provided by this product enables company admins to understand the demographic of the people following their page.
3. LinkedIn also heavily uses analytics internally to support critical requirements such as A/B testing.

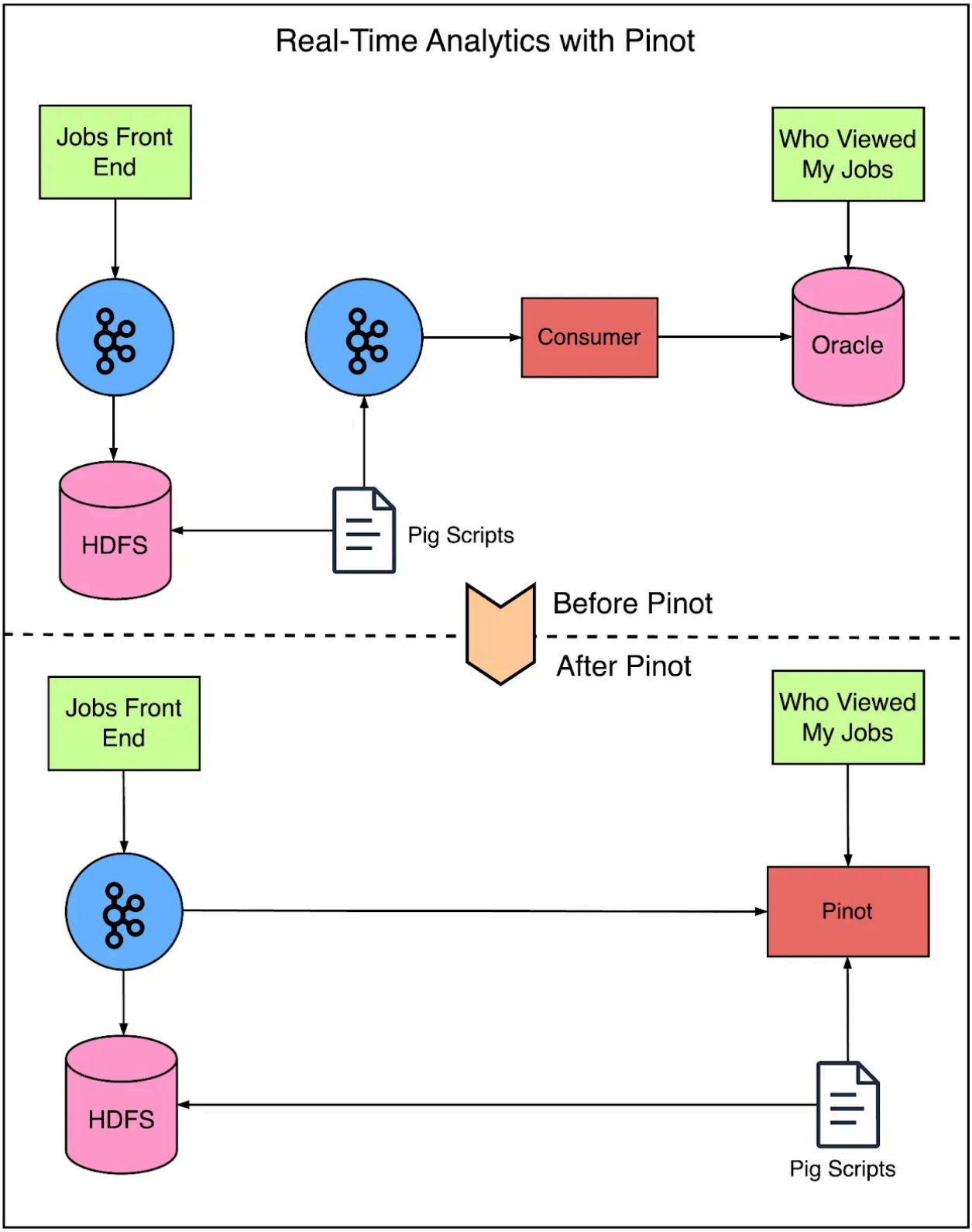
To support these key analytics products and many others at scale, the engineering team created Pinot.

Pinot is a web-scale real-time analytics engine designed and built at LinkedIn.

It allows them to slice, dice and scan through massive quantities of data coming from a wide variety of products in real-time.

But how does Pinot solve the problem?

The below diagram shows a comparison between the pre-Pinot and post-Pinot setup.



As you can see, Pinot supports real-time data indexing from Kafka and Hadoop, thereby simplifying the entire process.

Some of the other benefits of Pinot are as follows:

- Pinot supports low latency and high QPS OLAP queries. For example, it's capable of serving thousands of **Who's Viewed My Profile** requests while maintaining SLA in the order of 10s of milliseconds
- Pinot also simplifies operational aspects like cluster rebalancing, adding or removing nodes, and re-bootstrapping
- Lastly, Pinot has been future-proofed to handle new data dimensions without worrying about scale

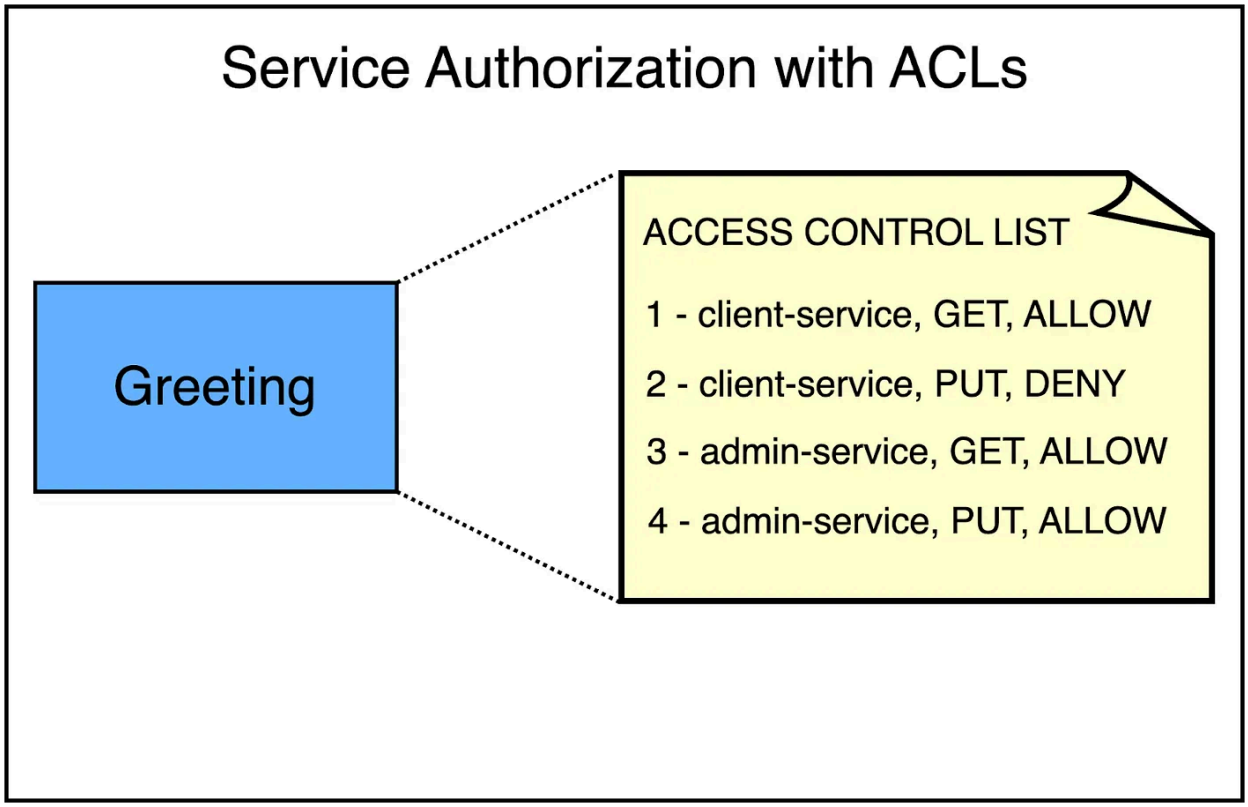
Authorization at LinkedIn Scale

Users entrust LinkedIn with their personal data and it was extremely important for them to maintain that trust.

After the SOA transformation, LinkedIn runs a microservice architecture where each microservice retrieves data from other sources and serves it to the clients. Their philosophy is that a microservice can only access data with a valid business use case. It prevents the unnecessary spreading of data and minimizes the damage if an internal service gets compromised.

A common industry solution to manage the authorization is to define Access Control Lists (ACLs). An ACL contains a list of entities that are either allowed or denied access to a particular resource.

For example, let's say there is a Rest.li resource to manage **greetings**. The ACL for this resource can look something like this.



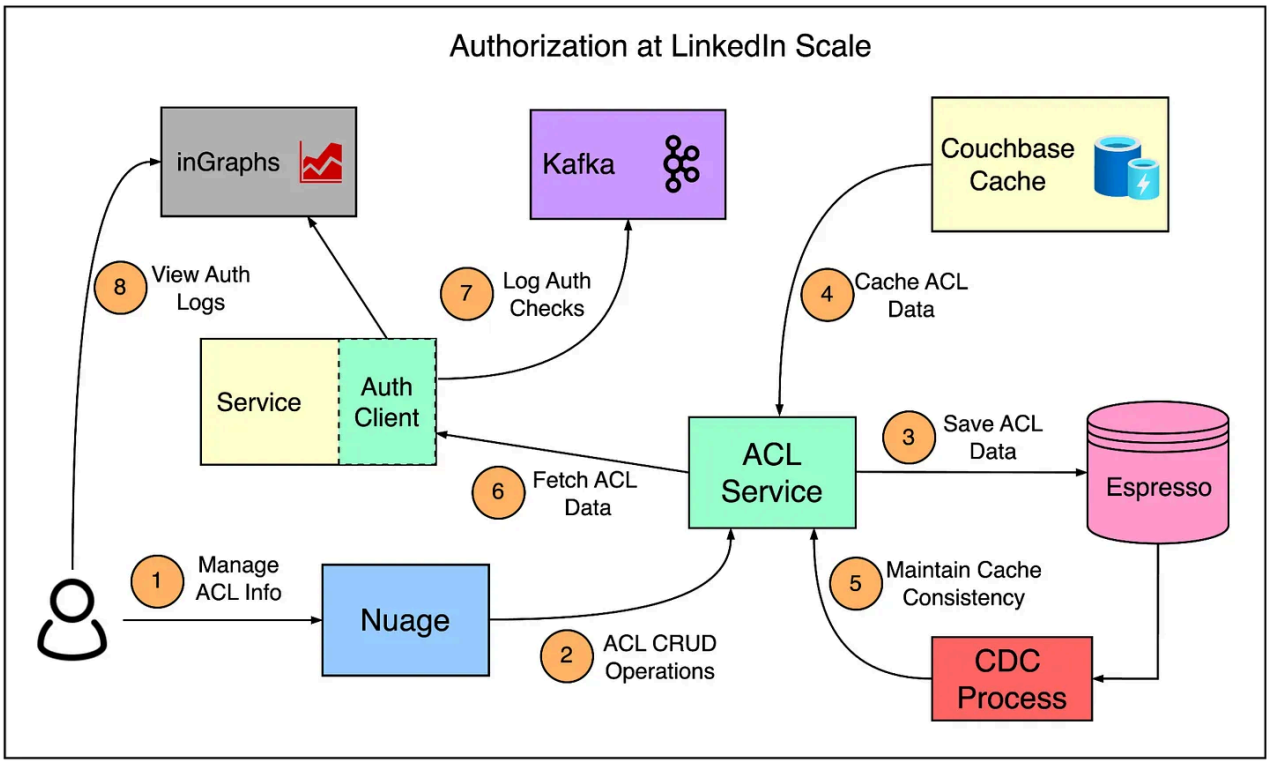
In this case, the **client-service** can read but not write whereas the **admin-service** can both read and write to greetings.

While the concept of an ACL-based authorization is quite simple, it's a challenge to maintain at scale. LinkedIn has over 700 services that communicate at an average rate of tens of millions of calls per second. Moreover, this figure is only growing.

Therefore, the team had to devise a solution to handle ACLs at scale. Mainly, there were three critical requirements:

- Check authorization quickly
- Deliver ACL changes quickly
- Track and manage a large number of ACLs

The below diagram shows a high-level view of how LinkedIn manages authorization between services.



Some key points to consider over here are as follows:

- To make authorization checks quick, they built an authorization client module that runs on every service at LinkedIn. This module decides whether an action should be allowed or denied. New services pick up this client by default as part of the basic service architecture.
- Latency is a critical factor during an authorization check and making a network call every time is not acceptable. Therefore, all relevant ACL data is kept in memory by the service.
- To keep the ACL data fresh, every client reaches out to the server at fixed intervals and updates its in-memory copy. This is done at a fast enough cadence for any ACL changes to be realized quickly.
- All ACLs are stored in LinkedIn's Espresso database. It's a fault-tolerant distributed NoSQL database that provides a simple interface.
- To manage latency and scalability, they also keep a Couchbase cache in front of Espresso. This means even on the server side, the data is served from memory. To deal with stale data in the Couchbase, they use a Change Data Capture system based on LinkedIn's Brooklin to notify the service when an ACL has changed so that the cache can be cleared.

- Every authorization check is logged in the background. This is necessary for debugging and traffic analysis. LinkedIn uses Kafka for asynchronous, high-scale logging. Engineers can check the data in a separate monitoring system known as **inGraphs**.

Conclusion

In this post, we've taken a brief look at the scaling journey of LinkedIn.

From its simple beginnings as a standalone monolithic system serving a few thousand users, LinkedIn has come a long way. It is one of the largest social networks in the world for professionals and companies, allowing seamless connection between individuals across the globe.

To support the growing demands, LinkedIn had to undertake bold transformations at multiple steps.

In the process, they've provided a lot of learnings for the wider developer community that can help you in your own projects.

References:

- A Brief History of Scaling LinkedIn
- Scaling LinkedIn's Edge with Azure Front Door
- How LinkedIn customizes Apache Kafka for 7 trillion messages per day
- Using Set Cover Algorithm to optimize query latency for a large-scale distributed graph
- Authorization at LinkedIn's Scale
- Rest.li: RESTful Service Architecture at Scale
- Real-time analytics at a massive scale with Pinot

SPONSOR US

Our newsletter puts your products and services directly in front of an audience that matters - hundreds of thousands of engineering leaders and senior engineers - who have influence over significant tech decisions and big purchases.

Space Fills Up Fast - Reserve Today

Ad spots typically sell out about 4 weeks in advance. To ensure your ad reaches this influential audience, reserve your space now by emailing **hi@bytebytego.com**

130 Likes · 11 Restacks

7 Comments

Write a comment...

Omri May 29

Some cool LinkedIn Lore is that Jay Kreps worked on implementing Kafka at LinkedIn, wrote a great blog post about it (https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying), and then went on to start Confluent, "the Kafka company".

LIKE (1) REPLY SHARE

Karen Brenchley An I Person May 29

Nice history of LinkedIn. The original version was designed so that in order to link with someone you didn't know, you needed to find someone who you knew and who also knew the potential connection. Then you'd ask the person you knew for an introduction,

and if the person agreed LI would connect you. That model didn't last long. (My internal LI user number is 1000.)

LIKE (1) REPLY SHARE

5 more comments...