That's a new one. A shift from David Patterson's "Datacenter as a computer" to "all global data centres as a computer"

DOI:10.1145/3701296

A look at Meta's planetary-scale computing infrastructure, including some key lessons from its development, as the company pursues its vision of "all global datacenters as a computer."

BY CHUNQIANG TANG

Meta's Hyperscale Infrastructure: Overview and Insights

Two reasons why they wrote this article:

(1) At times something useful at these large organisations trickles down to smaller organisations. Examples include: virtual memory originally introduced for IBM mainframes. More recent examples are Kubernetes and PyTorch.

HYPERSCALERS, SUCH AS Alibaba, Amazon, ByteDance, Google, Meta, Microsoft, and Tencent, have developed planetary-scale infrastructure to deliver cloud, Web, or mobile services to their global users. And though most practitioners may not directly build such hyperscale infrastructure, we believe it is beneficial to learn a bit about it. Historically, many widely used technologies have originated from advanced environments, including mainframes in the 1960s and hyperscale infrastructures in the past two decades. For instance, virtual memory had its origin in mainframes and is now common even in smartwatches. Similarly, Kubernetes and PyTorch originated in Google and Facebook, respectively, but have been adopted by organizations of all sizes. In addition to these specific technologies, the principles and lessons from hyperscale infrastructure may assist practitioners in building better systems in general.

This article provides a high-level overview of Meta's hyperscale infrastructure, focusing on key insights from its development, particularly in systems software. Where relevant, we highlight differences from public clouds, as varying constraints have led to distinct optimizations. Though much of the knowledge presented here has been shared and practiced within the industry and research community, including insights from our past publications, the article's primary contribution is to provide a holistic perspective that helps readers build a comprehensive mental model of hyperscale infrastructure end to end.

Engineering Culture

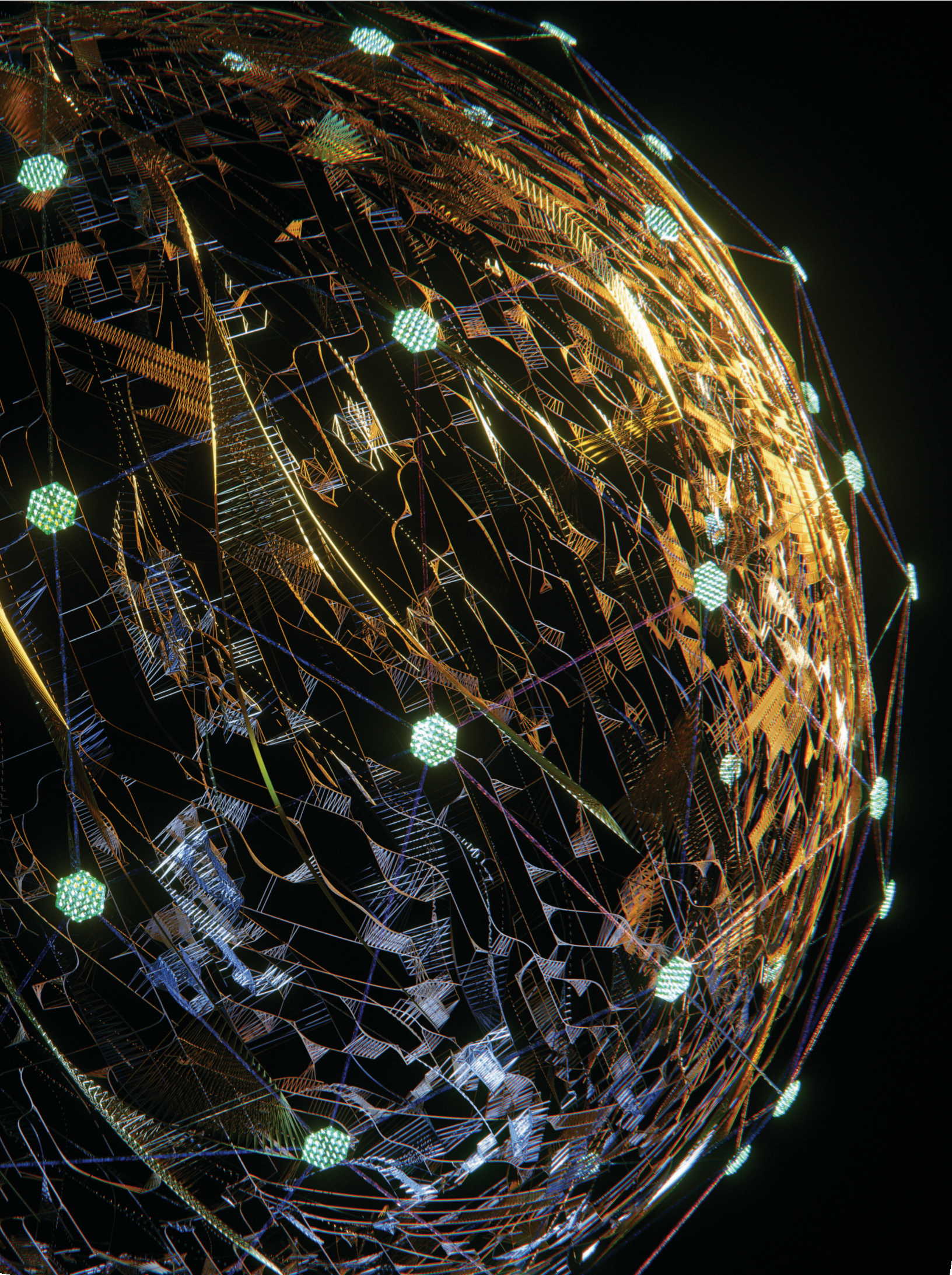
Before delving into the details of Meta's infrastructure, we first highlight several aspects of the company's engineering culture, because an organization's culture heavily influences its technology.

Move fast. Since its inception, Face-

» key insights

- **Meta's engineering culture emphasizes moving fast, technology openness, research in production, and shared infrastructure.**
- **To boost developer productivity, Meta has adopted continuous deployment universally and enabled more developers to write serverless functions rather than traditional service code.**
- **To reduce hardware costs, Meta utilizes hardware-software co-design at the datacenter scale and autonomously optimizes resource allocations, including workload migration, across global datacenters instead of limiting them to individual clusters.**
- **Meta's AI strategy involves co-designing the entire stack, from PyTorch to AI accelerators, networks, and ML models such as Llama.**

ILLUSTRATION BY PETER CROWTHER ASSOCIATES



book has ingrained and retained the "move-fast" culture, emphasizing agility and rapid iteration. This philosophy is evident in its strong commitment to continuous software deployment, which involves releasing the latest code into production as early as possible. Additionally, product engineers predominantly write code in stateless, serverless functions in PHP, Python, and Erlang for their benefits in simplicity, productivity, and iteration speed. Teams have the ability to quickly pivot their execution priorities without undergoing a lengthy replanning process, leaving ambiguous issues to be sorted out during iterative execution. This allows teams to quickly adapt and launch new products in response to evolving market conditions.

Technology openness. Meta champions technology openness, both internally and externally. Internally, we adopt the monorepo approach, storing the code for all projects in a single repository to facilitate code discovery and reuse, as well as cross-team contributions. While other organizations also use monorepos, they vary in the degree of openness. Some require designated owners for each project, with only these owners authorized to accept code changes, although others may propose changes. In contrast, with few exceptions, the vast majority of projects at Meta do not enforce such strict ownership rules. This openness encourages cross-team contributions and code reuse while discouraging the reinvention of similar technologies.

That is

At Meta, engineers directly commit code changes to the mainline of the monorepo, and software deployments are compiled from the mainline, that is, from the latest code, as opposed to some stable branches. For example, when a widely used library, such as the RPC library, is updated, the next release of every application dependent on this library will be automatically compiled with the latest version.

Externally, Meta's commitment to technology openness is demonstrated through its open-source hardware designs via the Open Compute Project[28] and open-source software projects such as PyTorch, Llama, Presto, RocksDB, and Cassandra. Also, much of Meta's infrastructure technology has been shared through research papers, with many examples in this article's references.

Unlike public clouds, which must provide various server types to accommodate diverse customer applications, Meta can optimize its applications to suit the hardware, thereby avoiding the proliferation of server types.

(1) An organisation's culture heavily influences its technology.

(A) Agility in coding solution and deploying it

(B) Meta has open sourced many software, many of its server designs, and many of its infrastructure design

(C) Research tackles the challenging problems in

Research in production. Meta's hyperscale infrastructure requires continuous innovation, but unlike most hyperscalers, the company does not have a dedicated systems research lab. Instead, all of its systems research papers are authored by teams developing production systems. These teams advance the state of the art while tackling challenging production issues at scale, then reflect on these experiences to distill working solutions into research papers. This approach ensures that the addressed problems are real and the solutions work at scale, aligning well with key criteria for successful systems research.

Common infrastructure. Some organizations empower individual teams to make local decisions about their technology stack. Meta, however, prioritizes standardization and global optimization. On the hardware side, servers supporting different products are all allocated from a shared server pool.[34] Moreover, for non-AI compute workloads, we offer only a single server type, equipped with one CPU and the same amount of DRAM (previously 64GB, now 256GB). Unlike public clouds, which must provide various server types to accommodate diverse customer applications, Meta can optimize its applications to suit the hardware, thereby avoiding the proliferation of server types.

Standardization also prevails on the software side. For instance, different Meta products previously used Cassandra, HBase, and ZippyDB[24] for key-value stores, but now all have converged to ZippyDB. Further, each common capability—such as software deployment,[19] configuration management,[33] service mesh,[31] pre-production performance testing,[11] in-production performance monitoring,[39] and in-production load testing[35]—is supported by a universally adopted tool.

Besides standardization, a key principle in achieving common infrastructure is our preference for reusable components over monolithic solutions. A good example of this is the component-reuse chain in our distributed file system, Tectonic.[29] Tectonic enhances scalability by using a distributed key-value store, ZippyDB,[24]

to store its metadata. ZippyDB further employs a common sharding framework, Shard Manager, to manage its data shards; Shard Manager, in turn, depends on Meta's mesh, ServiceRouter,[31] for shard discovery and request routing. Finally, ServiceRouter stores the service discovery and configuration data of the entire infrastructure, which is critical for the site's continuous operation, in the highly reliable, zero-dependency data store Delos.[3] Therefore, the component-reuse chain is Tectonic→ZippyDB→Shard Manager→ServiceRouter→Delos. All of these reusable components also serve many other use cases. In contrast, HDFS, a popular open source distributed file system, is a monolithic system that implements all of these components internally.

Culture case study: The Threads app. The development of the Threads app,[6] often compared to Twitter/X, exemplifies the aforementioned culture. Emphasizing moving fast, a small team developed Threads with just five months of technical work in a start-up-like environment. Moreover, once it was developed, the infrastructure teams were given only two day's notice to prepare for its production launch. Most large organizations would take longer than two days just to draft a project plan involving dozens of interdependent teams, let alone execute it. At Meta, however, we quickly established war rooms across distributed sites, bringing together both infrastructure and product teams to address issues in real time. Despite the tight timeline, the app's launch was highly successful, reaching 100 million users within just five days, making it the fastest-growing app in history.[6]

Common infrastructure was crucial for enabling teams to swiftly implement Threads and scale it reliably. Threads reused Instagram's Python backend as well as Meta's shared infrastructure components, such as the social-graph database,[5] key-value store,[24] serverless platform,[30] machine-learning (ML) training and inference platforms,[10] and configuration-management framework for mobile apps.[20]

Meta's internal technology openness, using a monorepo, allowed Threads to reuse some Instagram application code to accelerate its development. In terms of external technology openness, Threads aims to integrate with ActivityPub, the open social networking protocol, for interoperability with other apps. We have also publicly shared our experiences of rapidly developing Threads.[6]

Insight 1:
Despite many challenges, it is feasible for a large organization to maintain a culture of moving fast, using a common infrastructure, and sharing a monorepo without strictly enforcing code ownership.

End-to-End User Request Flow

We now dive into Meta's infrastructure technology. Meta products are supported by a shared service infrastructure. To provide a holistic view of this infrastructure, we explain how a user request is processed end-to-end, detailing all the components involved.

Request routing. *Dynamic DNS mapping.* When a user initiates a request to facebook.com, Meta's DNS server dynamically returns an IP address that is mapped to a Meta-operated small edge datacenter, known as point of presence (PoP), as depicted in Figure 1. This dynamic DNS mapping ensures that the chosen PoP is close to the user, while balancing load across PoPs. The user's TCP connection is terminated at the PoP, which maintains separate, long-lived TCP connections with Meta's datacenters. This split-TCP setup offers several advantages, including reduced TCP-establishment latency through the reuse of pre-established connections between PoPs and datacenters. A PoP typically has hundreds of servers but may have up to a few thousand. Hundreds of PoPs are positioned worldwide to ensure that most users have a PoP close to them, thereby ensuring short network latencies.

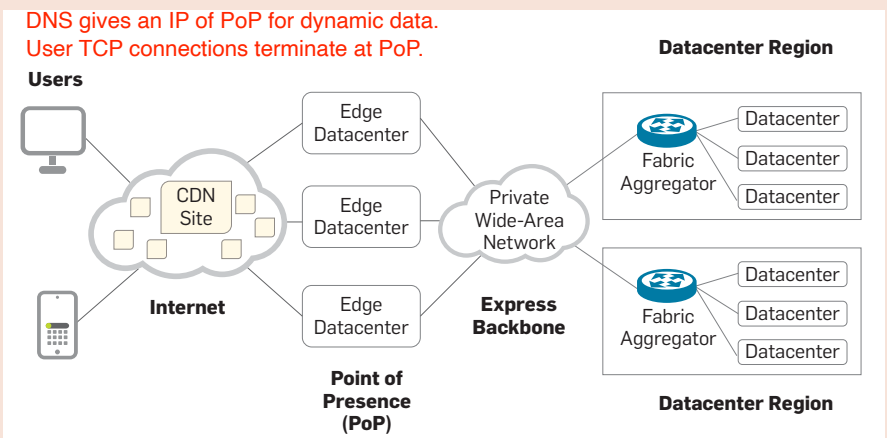
Static-content caching. If the user request is for static content, such as images and videos, it can be directly served at the PoP if the content is already cached there. Additionally, static content may be cached by the content delivery network (CDN), as shown in Figure 1. When a significant volume of Meta product traffic originates from an Internet service provider's (ISP's) network, Meta seeks to establish a mutually beneficial partnership by providing Meta Network Appliances to be hosted in the ISP's network to cache static content, thereby forming a CDN site. A CDN site typically has tens of servers, with some having more than a hundred. Thousands of CDN sites across the globe form our CDN for distributing static content.

Meta products use URL rewrites to redirect user requests to a nearby CDN site. When a Meta product provides a URL for a user to access static content, it rewrites the URL, for example, from `facebook.com/image.jpg` to `CDN109.meta.com/image.jpg`. If the image is not cached at CDN109 when the user requests it, CDN109 forwards the request to a nearby PoP. The PoP then forwards the request to the load balancer in a datacenter region, which retrieves the image from the storage system. On the return path, both the PoP and the CDN site cache the image for future use.

Dynamic-content request routing. If

Strateg

Figure 1. Meta's global infrastructure.



This picture is a good summary of this section.

the user request is for dynamic content such as a newsfeed, the PoP forwards it to a datacenter region. The selection of the target region is guided by a traffic-engineering tool[9] that periodically computes the optimal distribution of global traffic from PoPs to datacenters, considering factors such as datacenter capacity and network latency.

PoP-to-datacenter traffic travels through Meta's private wide-area network (WAN),[12] which globally interconnects Meta's PoPs and datacenters using optical fibers spanning tens of thousands of miles. Internal network traffic among our datacenters and PoPs significantly surpasses external-facing traffic between users and PoPs by several orders of magnitude, primarily due to data replication across datacenters and interactions among our microservices. The private WAN provides high bandwidth to serve this internal traffic.

Insight 2:
Meta's global infrastructure consists of CDN sites, edge datacenters, and main datacenters. Because of the high volume of our internal cross-datacenter traffic, we have built a private WAN to connect our datacenters, rather than relying on the public Internet.

Infrastructure topology. The table summarizes the aforementioned infrastructure components. Globally, there are tens of datacenter regions, hundreds of edge datacenters (PoPs), and thousands of CDN sites. Each datacenter region has multiple datacenters located within the radius of a few miles. Each datacenter uses up to a dozen main switchboards (MSBs) for power distribution, which also act as the primary sub-datacenter fault domains. An MSB failure can render 10 to 20 thousand servers unavailable.

Table. Number and size of infrastructure components.

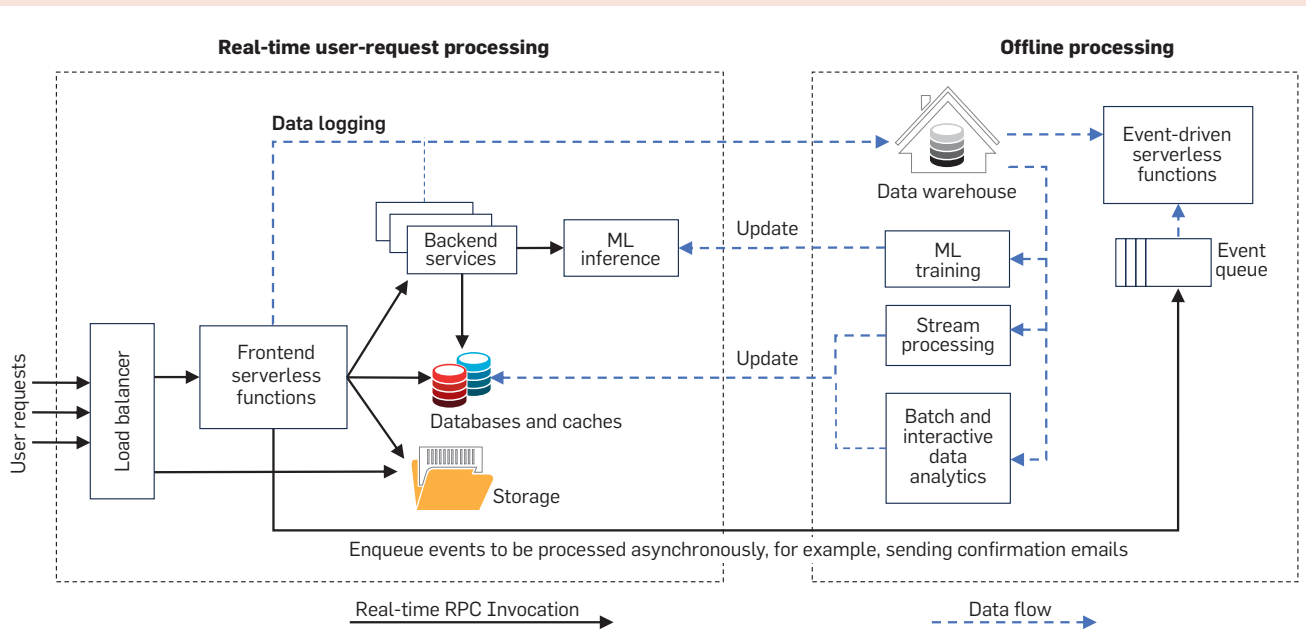
Entity type	Entity count	Servers in each entity
Region	O(10)	Up to one million
PoP	O(100)	Typically O(100) but up to O(1,000)
CDN site	O(1,000)	Typically O(10) but up to 100+
Datacenter	Multiple datacenters per region	O(100,000)
MSB	Up to a dozen MSBs per datacenter	Typically 10K to 20K

Edge network. A PoP is connected to multiple autonomous systems on the Internet and typically has multiple paths to reach a user network. When choosing a path between a PoP and a user, Border Gateway Protocol (BGP), by default, does not consider network capacity and performance. The PoP's network, however, takes these factors into consideration and advertises its preferred route to a network prefix.[32]

Datacenter network. Servers in a datacenter are interconnected by a datacenter fabric,[2] where network switches form a three-level Clos topology that can be scaled incrementally by adding more switches at the top level. With a sufficient number of top-level switches, the fabric can provide a non-blocking and non-oversubscribed network, enabling communication between any two servers at their full NIC bandwidth. We are moving toward eliminating network oversubscription within a datacenter.

Regional network. A fabric aggregator[14] connects datacenters within a region and further connects them to our private WAN. The fabric aggregator employs a topology akin to the fat tree, enabling the incremental addition of more switches to boost bandwidth. We aim to significantly reduce network oversubscription in a region so that cross-datacenter communi-

Figure 2. High-level architecture of software components running in a datacenter region. This is a highly simplified diagram, as Meta internally has O(10,000) backend services that exhibit a complex call graph.



cation within a region is not a bottleneck. This allows most services, except for ML training, to be scattered across datacenters in a region without worrying about a significant performance penalty.

Request processing. *Online processing.* When a user request reaches a datacenter region, it is processed along the path depicted in Figure 2. The load balancer spreads user requests across tens of thousands of servers that execute "frontend serverless functions." To process a user request, a frontend serverless function may invoke many backend services, some of which may further call "ML inference," for example, to retrieve recommendations for ads or newsfeed content.

During its execution, a frontend serverless function can enqueue events in the "event queue" for "event-driven serverless functions"[30] to process asynchronously. One such event could be sending a confirmation email after the user performs an action on the site. While frontend serverless functions directly affect user-perceived response time and hence have a tight latency service-level objective (SLO), event-driven serverless functions work asynchronously without affecting user-perceived response time, and are optimized for throughput and hardware utilization instead of latency. The ratio of servers executing frontend serverless functions to event-driven serverless functions is approximately 5:1.

Offline processing. The components on the right side of Figure 2 perform various offline processing to assist online processing on the left side. Decoupling online and offline processing enables independent optimization based on their respective workload characteristics. When handling user requests, frontend serverless functions and backend services log various types of data, such as ad-click-through or video-watch metrics, into the "data warehouse." This data feeds various offline processing. For instance, "ML training"[10] uses the data to update ML models, while "stream processing" can use the data to update the most-discussed topics on the site and store them in "databases and caches," which are then used during online user-request processing. Additionally, "batch analytics," powered by Spark and Presto, can periodically perform operations such as updating friend recommendations in response to new activities on the site. Finally, data updates in the data warehouse serve as a primary event source that triggers the execution of event-driven serverless functions.[30]

Aligning with the move-fast culture, we take continuous deployment of both code and configuration to extreme speeds and scales, enabling developers to quickly release new features and bug fixes, receive immediate feedback, and iterate rapidly.

> **Insight 3:**
> *Using a data warehouse as an intermediate layer to decouple online and offline processing simplifies the architecture and enables independent optimizations.*

Boosting Developer Productivity

A main purpose of a shared infrastructure is to boost developer productivity. While it is widely recognized that continuous software deployment and serverless functions can help make developers more productive, we have taken these approaches to the extreme.

Continuous deployment. Aligning with the move-fast culture, we take continuous deployment of both code and configuration to extreme speeds and scales, enabling developers to quickly release new features and bug fixes, receive immediate feedback, and iterate rapidly.

For configuration changes, our configuration-management tool[33] deploys more than 100,000 live changes daily in production, spanning O(10,000) services and millions of servers. These changes facilitate a variety of tasks, including load balancing,[9,31] feature rollouts, A/B tests, and overload protection.[25] At Meta, nearly every engineer who writes code also makes live configuration changes in production. Following the configuration-as-code paradigm, manual configuration changes undergo peer code review before being committed to a code repository. Once committed, these changes immediately enter the continuous deployment pipeline. Within seconds, the updated configuration can be pushed to potentially millions of subscribed Linux processes, triggering an upcall notification. The processes can immediately adjust their runtime behavior without restarts. In addition to manual changes, automation tools also drive configuration changes, for example, for load balancing.[9,31]

For code changes, our deployment tool[19] manages more than 30,000 pipelines to deploy software upgrades. At Meta, 97% of services adopt fully automated software deployments without any manual intervention: 55% utilize continuous deployment, instantly deploying every code change to production after passing automated tests, while the remaining 42% are automatically deployed on a fixed schedule, mostly daily or weekly. Take the frontend serverless functions in Figure 2 as an example. They run on more than half a million servers, with more than 10,000 product developers changing their code and thousands of code commits every workday. Despite this extremely dynamic environment, a new version of all serverless functions is released into production every three hours.

Even our network software is designed like regular services and optimized for frequent updates. For example, our private WAN[12] divides its network topology into multiple parallel planes, each responsible for a portion of the traffic and equipped with its own controller. This enables frequent updates of the controller software. Developers can experiment with new control algorithms by diverting traffic from one plane and deploying the new algorithm exclusively within that plane, without affecting other planes. Similarly, our network switch software[8] undergoes frequent updates, just like standard services. Leveraging the switch ASIC's "warm boot" feature, the data plane keeps forwarding traffic while the switch software undergoes an update.

Frequent code and configuration updates enable agile software development but increase the risk of site outages. To address this risk, we invest heavily in testing, staged rollouts, and health checks during updates.[19,33] Previously, we launched a company-wide campaign to boost code-deployment automation, increasing the adoption of fully automated code deployment guarded by health checks from 12% to 97%. Similarly, we implemented another initiative to ensure that all configuration changes undergo automated canary tests to uphold configuration safety. Overall, we find these investments in continuous deployment worthwhile, as it significantly boosts developer productivity.

Developers write FaaS code and leave it to the infrastructure to handle everything else through automation, including code deployment and auto-scaling in response to load changes.

Insight 4:
Even for a large organization with O(10,000) services, it is feasible to adopt continuous deployment at extreme scales and speeds. Specifically, 97% of our services adopt fully automated deployments without manual intervention, and 55% deploy every code change instantly.

Serverless functions. The widespread use of serverless functions (also known as function-as-a-service or FaaS) is another key driver that boosts developer productivity. Unlike traditional backend services, which can exhibit arbitrary complexity, FaaS is stateless and implements a simple function interface.[30] Each FaaS invocation is managed independently, with no side effects on other concurrent invocations, except through states stored in external databases. Due to its stateless nature, FaaS relies heavily on external caching systems[5,27] to achieve good performance when accessing databases.

Developers write FaaS code and leave it to the infrastructure to handle everything else through automation, including code deployment and auto-scaling in response to load changes. This simplicity allows Meta's more than 10,000 product developers to focus solely on product logic without concern for infrastructure management. Moreover, it prevents hardware waste caused by product developers over-provisioning resources.

Meta takes the usage of FaaS to the extreme to maximize developer productivity. Among O(10,000) engineers at Meta, the number of engineers writing FaaS code is about 50% greater than those writing code for regular services that they operate by themselves. This success is attributed not only to relieving product engineers of managing infrastructure but also to the usability of the integrated development environment (IDE) for FaaS. This IDE enables easy access to the social-graph database[5] and various backend systems through high-level language constructs. It also provides fast feedback through continuous integration tests.

As shown in Figure 2, Meta operates two FaaS platforms: one for frontend serverless functions and another for event-driven serverless functions. We refer to them as *FrontFaaS* and *XFaaS,*[30] respectively. FrontFaaS functions are written in PHP (we also have FaaS platforms for Python, Erlang, and Haskell functions). To support the high load generated by billions of users, we maintain over half a million servers that keep the PHP runtime running at all times. When a user request arrives, it is routed to one of these servers for immediate processing, without experiencing cold start time. When the site's load is low, we utilize auto-scaling to release some FrontFaaS servers for other services to use.

XFaaS shares many similarities with FrontFaaS, the key difference being that it executes non-user-facing functions that do not require subsecond response times but exhibit a highly spiky load pattern.[30] To avoid overprovisioning resources for peak loads, XFaaS employs a combination of optimizations to spread out function execution, including deferring the execution of delay-tolerant functions to off-peak hours, globally load-balancing function calls across regions, and implementing throttling based on quotas.

Product developers at Meta have been using FaaS as their primary coding paradigm since the late 2000s, even before the term *FaaS* became popular. Compared with serverless platforms in the industry, a unique aspect of our serverless platforms is that they allow multiple functions to execute concurrently in the same Linux process for higher hardware efficiency,[30] unlike public clouds that have to execute one function per virtual machine in order to ensure stronger isolation between different customers.

Insight 5:
Serverless functions have become the primary coding paradigm for product development at Meta. More than 10,000 Meta engineers write code for serverless functions, exceeding the number of engineers writing regular service code by 50%.

Reducing Hardware Costs

Besides boosting developer productivity, another main purpose of a shared infrastructure is to lower the cost of hardware. In this section, we highlight several examples of how software solutions help reduce hardware costs.

All global datacenters as a computer. Most infrastructures place the burden of managing the complexities of geo-distributed datacenters on users, requiring them to manually determine the number of replicas for their services and select the regions for deployment, all while ensuring that service-level objectives are met. This complexity often leads to hardware wastage due to overprovisioning, uneven load distribution across regions, and insufficient cross-region migration to adapt to changes in workload demand and datacenter supply.

In contrast, Meta is evolving from the practice of "the datacenter as a computer"[4] (DaaC) to the vision of "all global datacenters as a computer" (Global-DaaC).[40] With Global-DaaC, users simply request the global deployment of a service, leaving the infrastructure to manage all the details: determining the optimal number of service replicas, placing these replicas across datacenter regions based on service-level objectives and available hardware, selecting the best-matching hardware type, optimizing traffic routing, and continuously adapting service placement in response to workload changes. Compared with public clouds, Meta can more easily realize Global-DaaC because it owns all its applications and can move them across regions as needed; public clouds lack this flexibility with their customers' applications.

To implement Global-DaaC, our tools seamlessly coordinate resource allocation across all levels: global, regional, and within individual servers. First, our global capacity-management tool[13] uses RPC tracing to identify service dependencies and construct resource-consumption models, then employs mixed-integer programming to break down a service's global capacity needs into regional quotas. Next, our regional capacity-management tool[26] assigns server resources to these regional quotas to form virtual clusters. Unlike physical clusters, a virtual cluster can comprise servers from different datacenters in the same region, and its size may dynamically grow or shrink. During runtime, our container-management tool[34] allocates containers in these virtual clusters, often spreading a job's containers across multiple datacenters in the same region for improved fault tolerance. Finally, at the server level, our kernel mechanisms[21,37] ensure proper sharing and isolation of memory and I/O resources allocated to individual containers.

Stateful services, such as databases, benefit from Global-DaaC. These services are typically sharded, with each container hosting multiple data shards for efficiency. Our global service placer (GSP) uses constrained optimization to determine the optimal number of replicas for each data shard and their placement across regions. Then, our sharding framework[23] works within the constraints set by GSP to allocate shard replicas to containers and dynamically migrate them in response to load changes.

Similarly, ML workloads benefit from Global-DaaC. For ML inference, models are managed similarly to data shards, with the number of model replicas and their locations determined by GSP. For ML training, it requires the collocation of training data and GPUs in the same datacenter region. Each team receives a global GPU capacity quota and submits training jobs to a global job queue. Our ML training scheduler[10] automatically selects regions for data replication and GPU allocation to ensure the colocation of data and GPUs while maximizing GPU utilization.

Insight 6:
Meta is evolving from the practice of "the datacenter as a computer"[4] to the vision of "all global datacenters as a computer."[40] In this model, the infrastructure autonomously determines and migrates deployments across global datacenters in response to workload changes, eliminating the need for user involvement. We have successfully demonstrated this approach for databases, ML systems, and diverse services operating at the scale of O(100,000) servers and O(100,000) GPUs.

Hardware and software co-design. While hardware and software co-de-

sign within a single server is common, we have elevated it to the global scale to use software solutions to overcome the limitations of lower-cost hardware.

Low-cost fault tolerance. Public clouds tend to provide hardware with higher availability because their customers' applications might not be sufficiently fault tolerant. In contrast, since all our applications are under our control, we can ensure they are implemented in a fault-tolerant manner to run on cheaper hardware with lower availability guarantees. For example, a server rack in public clouds may use dual power supplies and dual top-of-rack (ToR) switches to ensure high availability and facilitate switch maintenance without disrupting running workloads. In contrast, our racks have neither dual power supplies nor dual ToR switches. Instead, hardware redundancies occur only at the much larger scope of the power main switchboards (MSBs), each covering about 10,000 to 20,000 servers. For every six MSBs, there is only one reserve MSB as a backup. Moreover, virtual machines (VMs) in public clouds often use network-attached block devices, which enable live VM migration. In contrast, our containers use low-cost, directly attached SSDs for root disks, which hinders live-container migration during datacenter maintenance operations.

We use software solutions to overcome the limitations of lower-cost hardware. First, our resource-allocation tools[23,26,34] ensure that a service's containers and data shards are sufficiently spread across different sub-datacenter fault domains (MSBs) for better fault tolerance. Second, through a cooperative protocol that allows a service to weigh in on the lifecycle management of its containers,[19] we ensure that maintenance operations respect application-level constraints, such as avoiding simultaneous shutdowns of two replicas of the same data shard. Finally, Global-DaaC ensures that services are deployed to withstand the simultaneous loss of an entire datacenter region, one MSB in each region, and a certain percentage of random servers in each region. We routinely conduct tests in production to ensure that these properties hold so our services are fault tolerant.[36]

Global-DaaC ensures that services are deployed to withstand the simultaneous loss of an entire datacenter region, one MSB in each region, and a certain percentage of random servers in each region.

Though our infrastructure is designed to withstand the loss of an entire datacenter region without affecting users, the increasing number of regions has raised the possibility of two nearby regions being simultaneously affected by a large-scale natural disaster such as a hurricane. Instead of over-provisioning capacity to tolerate the simultaneous loss of two regions, we employ a software-based approach[25] that, in the event of losing multiple regions, deactivates less-critical product features and gracefully degrades service quality, such as delivering lower-quality videos, to reduce the load.

Eliminating the costs of routing proxies. Unlike traditional service meshes that predominantly use sidecar proxies to route RPC requests, Meta's service mesh[31] uses proxies to route only 1% of RPC requests across our fleet. The remaining 99% use a routing library linked into service executables for direct client-to-server routing, bypassing intermediate proxies. While this unconventional approach saves us O(100,000) servers needed for proxies, it introduces deployment challenges due to the library being compiled into around O(10,000) services, each with its own deployment schedule. Our software deployment and configuration-management tools[19,33] help make these challenges manageable.

Tiered storage and local SSDs. Based on access frequency and latency tolerance, we categorize data as hot, warm, or cold, with each category using a different storage system to optimize cost-effectiveness. Hot databases and caches, such as the social graph database,[5] store data in memory and solid state drives (SSDs).

Warm data, including videos, images, and data in the data warehouse (for example, user activity logs), is stored in a distributed file system[29] that utilizes hard disk drives (HDDs) to store data. Each storage server is equipped with one CPU, 36 HDDs, and two SSDs for metadata cache.

For rarely accessed cold data, such as a decade-old high-resolution video, we archive them with high-density HDD servers, each with one CPU and 216 HDDs, which provides a good balance between total cost of ownership and data-restoration speed. These

HDDs are powered off most of the time, as they are not in active use.

Among workloads that store data on SSDs, some can tolerate longer-tail latencies and opt for SSD-based shared remote storage for better SSD utilization. However, workloads with strict latency requirements still use directly attached local SSDs. Compared with other hyperscale infrastructures, we more frequently employ local SSDs to reduce costs, despite the management complexities involved. For instance, imbalanced load distribution can lead to the underutilization and stranding of local SSDs. Additionally, failure recovery is complicated by data becoming trapped in the SSDs of failed servers. To address these challenges, we use our common sharding framework[23] to implement stateful services with local SSDs, solving the issues once and reusing the solution across many services.

Insight 7:
To reduce hardware costs, we use software solutions to overcome the limitations of lower-cost hardware. Although this approach adds complexity to the software stack, we consider the trade-off worthwhile due to the significant cost savings.

In-house hardware design. We design our own datacenters[17] and hardware—servers, network switches, video accelerators, and AI chips[15]—for better costs and power efficiency. In datacenters, power is the most constrained resource because it is fixed at the time of datacenter construction and hard to expand later during a datacenter's 20-to-30-year lifespan. In contrast, the network and servers can be upgraded as needed. Power in a datacenter is often oversubscribed. To prevent over-drawing power when workloads surge, an automation tool[38] coordinates power-capping actions across the power-delivery hierarchy.

Our hardware designs often achieve cost and power savings through hardware/software co-design (for example, optimizing SRAM usage in our AI chip based on our workloads[15]), and by removing components unnecessary to us (for example, eliminating compressor-cooled air conditioning[17]). Additionally, in-house development of network switches and their companion software[8] enables us to treat switch software like a regular service and deploy updates frequently. Most of our hardware designs are open source through the Open Compute Project.[28]

Insight 8:
To reduce hardware costs and power consumption, Meta designs its own datacenters, servers, racks, and network switches, and shares these designs through open source.

Designing Scalable Systems

A recurring theme in hyperscale infrastructure is the design of scalable systems. Decentralized systems designed for the Internet environment, such as BGP, BitTorrent, and distributed hash tables (DHTs), are often lauded for their scalability. However, in a datacenter environment, which is less resource constrained and under the control of a single organization, our experiences indicate that centralized controllers not only achieve ample scalability but also are simpler and can make higher-quality decisions.

Deprecating decentralized controllers. In this section, we discuss several examples of the trade-off between centralized and decentralized controllers. For network switches in our datacenter fabric, although they still use BGP for compatibility, the fabric has a centralized controller capable of overriding routing paths during network congestion or link failures.[1]

Except for BGP, we have migrated almost all decentralized controllers to centralized ones. For example, in our private WAN,[12] we transitioned from decentralized RSVP-TE to a centralized controller to compute preferred traffic paths and proactively establish backup paths for common failure scenarios. This has resulted in more efficient network resource usage and faster convergence during network failures.

For key-value stores, DHTs use multi-hop routing to determine the server responsible for a given key, while Cassandra uses consistent hashing for this purpose. Both function without a central controller. In contrast, to achieve better load balance, our sharding framework[23] uses a central controller to dynamically reassign key-encapsulating shards to servers.

For bulk-data distribution, we transitioned from BitTorrent to Owl,[16] which centralizes the decision of where a peer should fetch data, resulting in significantly faster download speeds. Note that both Owl and our private WAN[12] centralize the control plane for better decision making but still use a decentralized data plane for actual data forwarding or downloading.

For small-metadata distribution (further explained in Figure 4), we initially used a three-level distribution tree implemented in Java. The tree's intermediate nodes were dedicated proxy servers, and its leaf nodes were application subscribers that could dynamically join and leave. When this implementation could not scale further, we transitioned to a peer-to-peer distribution tree, where intermediate nodes were also application subscribers that forwarded data to other subscribers. Among millions of application subscribers, however, a subset often experienced noisy performance issues due to their non-dedicated nature. Consequently, using them as intermediate nodes to forward traffic was less reliable, leading to frequent and time-consuming debugging. Eventually, after a few years of production use, we abandoned the peer-to-peer distribution tree and reverted to the original architecture that uses dedicated proxy servers. We replaced

Figure 3. Sidecar-proxy-based service mesh.

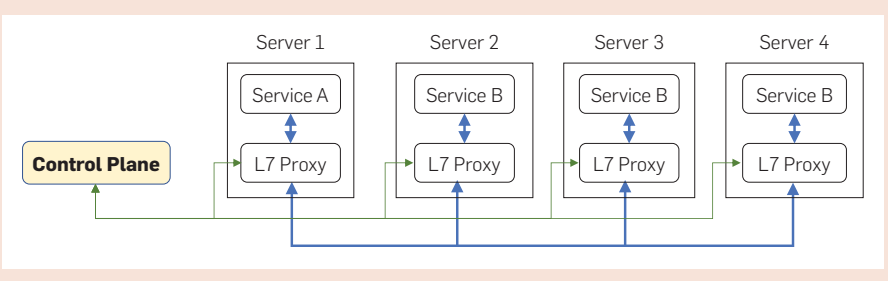
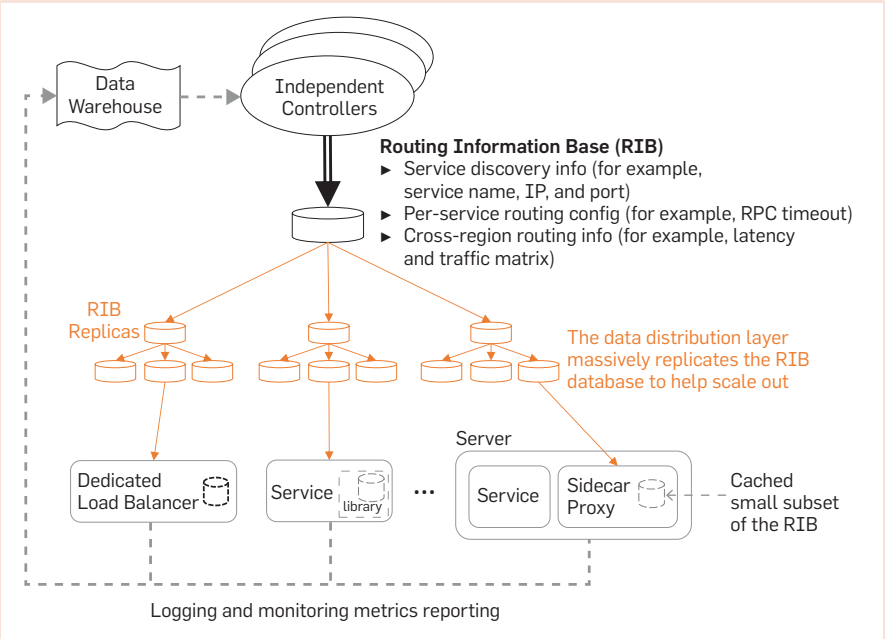


Figure 4. ServiceRouter's scalable service-mesh architecture.



the original Java implementation with a more performant C++ implementation, which scaled well to tens of millions of subscribers.

Insight 9:

In a datacenter environment, we prefer centralized controllers over decentralized ones due to their simplicity and ability to make higher-quality decisions. In many cases, a hybrid approach—a centralized control plane combined with a decentralized data plane—provides the best of both worlds.

Case study: Scalable service mesh. In this section, we use Meta's service mesh, ServiceRouter,[31] as a case study to illustrate the design of scalable systems and demonstrate that centralized controllers combined with a decentralized data plane can scale well in a datacenter environment. ServiceRouter routes billions of RPCs per second across millions of layer-7 (L7, that is, application layer) routers.

Figure 3 depicts a commonly used service mesh in the industry, where each service process is accompanied by an L7 sidecar proxy that routes RPCs for the service. For example, when service A on server 1 sends requests to service B, the proxy on server 1 load balances them across servers 2, 3, and 4. While this solution is widely adopted, it is not scalable for hyperscale infrastructure because the central controller cannot scale to directly configure the routing tables of millions of sidecar proxies. The central controller has a dual function of generating global routing metadata and managing each L7 router. To scale out, we keep the former in the central controller but transfer the latter to L7 routers, making each L7 router self-configuring and self-managing.

Figure 4 illustrates the scalable architecture of ServiceRouter. At the top, different controllers independently execute distinct functions such as registering services, updating measured network latencies, and computing a per-service cross-region routing table. Each controller independently updates the central routing information base (RIB) and is not concerned with configuring or managing individual L7 routers. The RIB is a Paxos-based database[3] and can scale out through sharding. With the help of the RIB, the controllers become stateless and can easily scale out through sharding as well. For example, multiple controller instances can concurrently compute cross-region routing tables for different services.

In the middle of Figure 4, the distribution layer leverages thousands of RIB replicas to handle read traffic from millions of L7 routers. At the bottom, guided by the RIB, each L7 router self-configures without the direct involvement of the control plane. Heterogeneous L7 routers are supported, which can be load balancers, services with embedded routing libraries, or sidecar proxies.

As ServiceRouter shows, we can achieve good scalability with centralized controllers through techniques like stateless controllers, controller sharding, and removing non-essential functions, such as managing individual L7 routers, from central controllers.

Future Directions

Despite the complexity of Meta's hyperscale infrastructure, here we provided a concise, high-level overview, emphasizing key insights from its development. To conclude, we share our thoughts on potential future trends for hyperscale infrastructure.

AI. AI workloads have become the single largest category of workload in datacenters. We anticipate that, before the end of this decade, more than half of the power in datacenters will be dedicated to AI workloads. Due to its distinct characteristics, such as being more resource-intensive and requiring higher-bandwidth networks, AI is expected to profoundly reshape every aspect of infrastructure. In the past two decades, hyperscale infrastructures have succeeded mostly by taking the *scaling-out* approach to utilize a large number of low-cost commodity servers. Future AI clusters, however, will more likely take the *scale-up* approach used by past supercomputers, such as using remote direct memory access (RDMA) over Ethernet to provide the high-bandwidth, low-latency network required for large-scale ML training.[18] Meta's approach to AI is distinguished by co-designing the full stack, from PyTorch to ML models, AI chips, networks, datacenters, servers, storage, power, and cooling.

Domain-specific hardware. Reversing the trend of diminishing hardware diversity in the 2000s, we anticipate a proliferation of custom and specialized hardware for various purposes, such as AI training and inference, virtualization, video encoding, encryption, compression, tiered memory, as well as in-network and in-storage processing. This is because economies of scale allow hyperscalers to design and deploy specialized hardware in large quantities to reduce costs. Consequently, this will pose challenges for the software