How Google Actually Works: A Systems Architecture of Planet-Scale Computing

Harsh R. Bagtharia[1,2]

[1] *B.Tech, Malla Reddy College of Engineering*

[2] *Chief Technology Officer (CTO), Tenexis*

August 2026

Abstract

Google's consumer and cloud products appear to users as discrete applications: Search, Gmail, YouTube, Drive, Gemini, and Google Cloud. At infrastructure level, however, these products are built on a large collection of distributed systems that coordinate computation, storage, networking, identity, scheduling, and specialized hardware across geographically distributed facilities. The central engineering problem is not merely to increase computational capacity, but to provide low latency, high availability, scalable storage, and predictable behavior while operating over large fleets of hardware in which individual failures are routine rather than exceptional.

This paper synthesizes four supplied research documents into a single architectural model of Google's large-scale computing environment. The analysis combines historical Google systems research with the more recent public architecture and engineering material contained in the source documents. The resulting model spans the user-facing edge, software load balancing, service front ends, cluster scheduling, distributed databases, distributed storage, data-center and wide-area networking, cloud network virtualization, and artificial-intelligence accelerators. Particular attention is given to the evolution from GFS to Colossus, from ad hoc cluster management to Borg, from sharded database systems to Spanner, from hardware load balancers to Maglev, and from conventional networking to software-defined fabrics such as B4 and Jupiter.

The synthesis identifies five recurring architectural principles: **horizontal scaling, abstraction over physical resources, separation of compute/storage/networking, failure-aware distributed coordination, and workload-specific optimization**. It also shows that Google's architecture is not a single static stack. Historical systems remain important as conceptual foundations, while current public materials describe newer generations and extensions. Claims about exact production topology, server counts, Search internals, Gemini serving, and proprietary control policies remain necessarily limited because Google does not publicly expose a complete current production blueprint.

The paper concludes that Google's defining infrastructure achievement is not the construction of one extraordinarily powerful computer. It is the construction of a software-controlled environment in which very large numbers of imperfect machines, storage devices, network links, and accelerators can be presented to applications as coherent pools of resources.

Keywords: Google infrastructure, warehouse-scale computing, distributed systems, Borg, Colossus, Bigtable, Spanner, Maglev, Jupiter, B4, Andromeda, TPU, cloud architecture.

Contents

1 Introduction

A Google query is presented as a simple interaction. A user enters text, presses Enter, and receives a response. The visible interaction conceals a much larger computational process. The request may enter Google's network through a geographically distributed edge, pass through traffic-management and front-end infrastructure, invoke multiple application services, retrieve state from distributed databases and storage systems, and use specialized accelerators for machine-learning workloads. The response is then assembled and returned through the network to the user.

This complexity is a direct consequence of scale.

A conventional application can often be represented as:

The model becomes increasingly inadequate as demand grows. Capacity must be distributed across machines, state must be partitioned and replicated, traffic must be balanced across many endpoints, and workloads must be scheduled across heterogeneous resources. Network latency becomes significant, and component failure becomes inevitable. A system containing thousands or tens of thousands of machines cannot treat hardware failure as an exceptional event; it must assume that some component is unavailable at almost any given moment.

Google's infrastructure research is important because it demonstrates a systematic response to these constraints. The Google File System (GFS) addressed large-scale storage over commodity hardware. MapReduce demonstrated large-scale parallel data processing. Bigtable introduced a distributed structured-data system. Borg converted a large fleet into a schedulable compute resource. Spanner addressed globally distributed transactions and external consistency. Maglev moved network load balancing into software. B4 and Jupiter demonstrated software-defined control of wide-area and data-center networks. Colossus evolved Google's distributed storage architecture beyond the limitations of the original GFS design. TPUs extended the same philosophy into specialized AI computation.

The common theme is that Google does not simply make individual components larger. Instead, it builds **abstractions that allow many components to function as one logical system**.

The central question of this paper is therefore:

> **How does Google architect a computing platform capable of operating at planetary scale despite continuous hardware, network, software, and workload variability?**

The paper examines this question through an integrated systems view rather than through a product-by-product description.

2 Objectives and Research Questions

2.1 Objective

The objective is to construct a coherent public, high-level map of Google's large-scale computing architecture from the user-facing edge to physical infrastructure, while distinguishing historical systems, current publicly documented systems, architectural inference, and proprietary implementation details.

2.2 Research Questions

The synthesis addresses the following questions:

1. Why do traditional centralized architectures become inadequate at Google's scale?
2. Why is horizontal scaling fundamental to Google's infrastructure?

3. How are compute resources abstracted away from individual physical machines?
4. How is data stored across large populations of disks and servers?
5. How are structured data and transactions distributed?
6. How are user and service traffic distributed across large fleets?
7. How is Google's data-center network scaled without making the network a central bottleneck?
8. How are workloads scheduled and recovered from machine failures?
9. How are consistency and coordination achieved across geographic regions?
10. How are AI workloads changing the relationships among compute, networking, and storage?
11. What architectural trade-offs follow from these choices?
12. Which parts of Google's current production architecture can be established from public evidence, and which remain unknown?

3 Methodology

This paper is an **integrative technical synthesis**, not a statistical meta-analysis.

Four supplied Markdown documents were treated as the source corpus:

1. A deep-research dossier describing the conceptual map, detailed component behavior, technical terminology, failure analysis, scale claims, and open questions.
2. A complete research-paper draft organized around the historical and architectural evolution of Google's systems.
3. An updated research-paper iteration incorporating later public material and newer systems information.
4. A generic synthesis template intended to guide comparison, methodology, discussion, limitations, and research-paper organization.

The first three files contain the substantive Google-specific material. The fourth file provides methodological guidance for integrating multiple drafts, but its example studies, sample sizes, statistical tests, forest plots, and placeholder effect estimates are unrelated to Google's infrastructure and are therefore not treated as factual evidence.

The synthesis followed four editorial operations:

3.1 Evidence consolidation

Overlapping descriptions of the same system were combined into a single account. Historical and current descriptions were retained where they provide different evidence about architectural evolution.

3.2 Contradiction and scope control

Claims that could be mistaken for an exact description of Google's present production environment were narrowed to the level supported by the supplied documents. For example, historical Search components such as SuperRoot, Ascorer, and Twiddlers are presented as examples of publicly discussed or source-derived Search architecture rather than as a definitive map of all current Search production systems.

3.3 Temporal qualification

The paper distinguishes among:

- **historically documented systems**, such as GFS and the original Borg publication;
- **current publicly described systems**, such as Colossus and contemporary TPU infrastructure;
- **architectural inference**, where the documents combine published system behavior into a high-level request flow; and
- **proprietary or unknown details**, which cannot be established from public documentation.

4 The Problem: Computing at Planetary Scale

4.1 The Limits of the Single-Machine Model

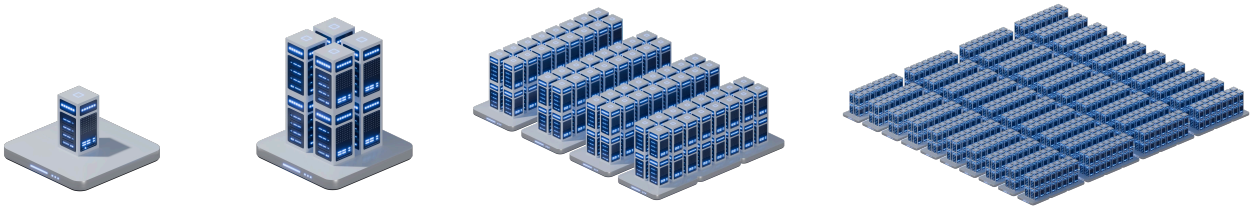
A single machine has finite CPU capacity, memory, storage, I/O throughput, and network bandwidth. Vertical scaling can increase those resources, but eventually produces economic, physical, and reliability constraints.

The more consequential limitation is that a single machine creates a narrow failure domain.

If the machine fails:

A large service cannot rely on this model.

The natural response is horizontal scaling:



But horizontal scaling changes the problem. The system must now coordinate machines that do not share memory, communicate through networks, experience different load levels, and fail independently.

4.2 Failure Becomes Normal

At small scale, a machine failure can be treated as an incident.

At hyperscale, failure becomes part of normal operation.

A system must be able to answer:

- What happens when a worker disappears?
- What happens when a storage device becomes unreadable?
- What happens when a network link fails?
- What happens when a replica becomes unreachable?
- What happens when a request becomes a straggler?
- What happens when a software deployment introduces an error?

Reliability therefore comes from **detection, redundancy, recovery, and abstraction**, not from the assumption that every physical component is reliable.

A useful abstraction is:

$$\text{Imperfect components} + \text{Failure detection} + \text{Redundancy} + \text{Recovery} = \text{Reliable service}$$

4.3 Coordination Becomes a First-Class Problem

Once state is distributed, correctness is no longer purely local. Network latency, message loss, clock uncertainty, replica divergence, and partial failure all affect system behavior.

The architecture must therefore solve two problems simultaneously:

`How do we scale?`

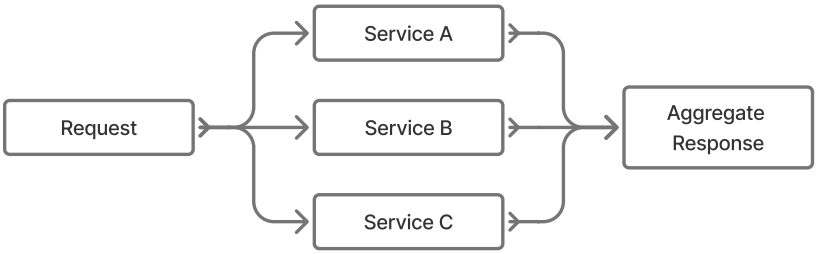
and:

`How do we remain correct while scaling?`

This is the central tension of distributed systems.

4.4 Tail Latency

Distributed services often fan a single request out to multiple machines:



If one component is substantially slower than the others, the aggregate operation may wait for it.

As fan-out increases, rare slowdowns become increasingly relevant. The mean latency of individual components can therefore be much less informative than high-percentile or **tail latency**.

Dean and Barroso's *The Tail at Scale* established this as a fundamental performance concern for large distributed services.

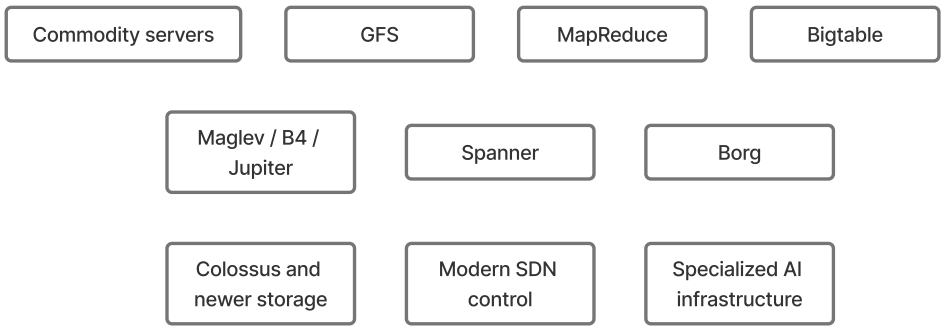
Thus, Google's infrastructure is not merely optimized for throughput. It is also designed around the question:

> **How can the system keep the slowest relevant component from determining the user's experience?**

5 Historical Evolution of Google's Infrastructure

Google's architecture was not designed as a single coherent system from the beginning. It evolved through successive generations as existing abstractions became inadequate.

A simplified progression is:



The systems overlap chronologically. The sequence is therefore best interpreted as an evolution of major infrastructure ideas rather than a strict replacement chain.

5.1 From Enterprise Hardware to Software Abstraction

Traditional enterprise systems often relied on highly specialized hardware for storage, load balancing, and networking. The Google approach shifted the reliability burden toward software and fleet-level orchestration.

The key idea was:

> **Cheap, replaceable hardware can become the foundation of a reliable service if the software is designed to expect failure.**

5.2 GFS and Distributed Storage

GFS demonstrated that a filesystem could distribute large files across many machines and disks while tolerating hardware failure.

Its design targeted:

- large data sets;
- high aggregate throughput;
- large files;
- commodity hardware;
- fault tolerance.

GFS was historically transformative, but its architecture also exposed limitations as Google's workloads evolved. In particular, centralized metadata management became increasingly problematic for much larger and more diverse workloads.

5.3 MapReduce and Large-Scale Computation

MapReduce provided an abstraction for processing large data sets across many machines.

Its significance was broader than the individual programming model. It demonstrated that application developers could reason about a distributed computation through a higher-level abstraction while the infrastructure managed placement, parallel execution, data movement, and recovery.

5.4 Bigtable and Structured Data

Bigtable extended the abstraction from files to structured data.

Instead of directly managing physical machines, an application could access a logical table whose underlying data was partitioned across a distributed fleet.

5.5 Borg and Fleet-Scale Scheduling

Borg addressed a different bottleneck: how to decide where applications should run.

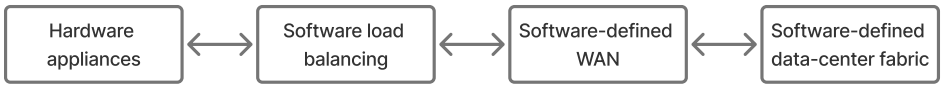
The fleet itself became a resource pool.

5.6 Spanner and Global State

Spanner addressed the challenge of coordinating strongly consistent transactions over geographically distributed replicas.

5.7 Maglev, B4, and Jupiter

The networking architecture evolved in parallel:



The network increasingly became something the system programmed rather than merely something the system used.

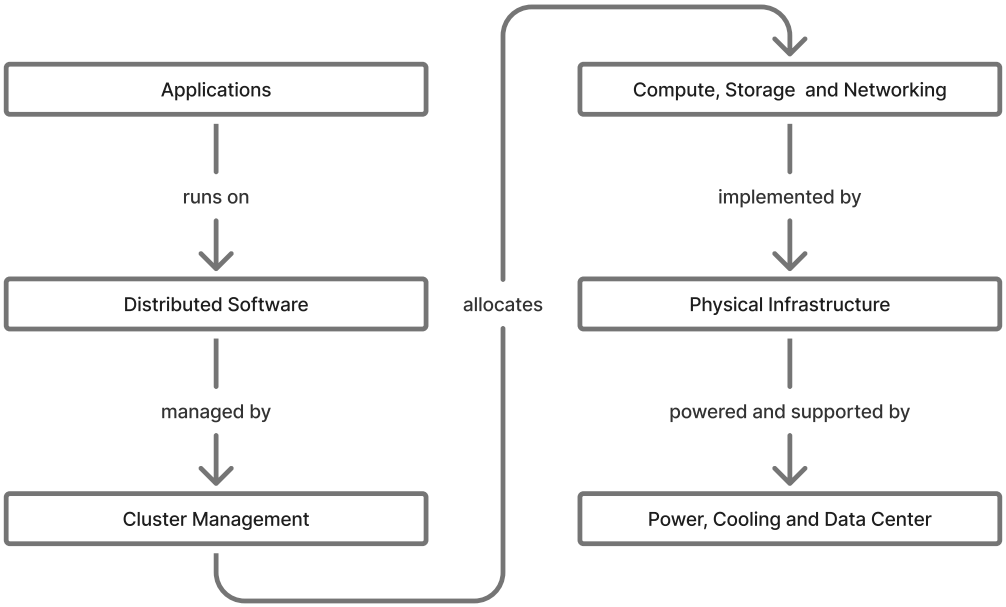
5.8 Colossus and Modern Storage

Colossus continued the storage lineage of GFS while addressing greater scale, metadata distribution, workload diversity, and tighter integration with modern cloud and AI workloads.

6 The Warehouse-Scale Computer

Google's warehouse-scale-computing literature established a useful mental model: a data center is not merely a building containing independent servers. It can be treated as an integrated computing platform whose hardware and software are designed together.

The abstraction becomes:



In this model, the individual server is a component, not the fundamental unit of the architecture.

This changes how the entire system is engineered.

A machine is permitted to fail because the service is not conceptually bound to that machine.

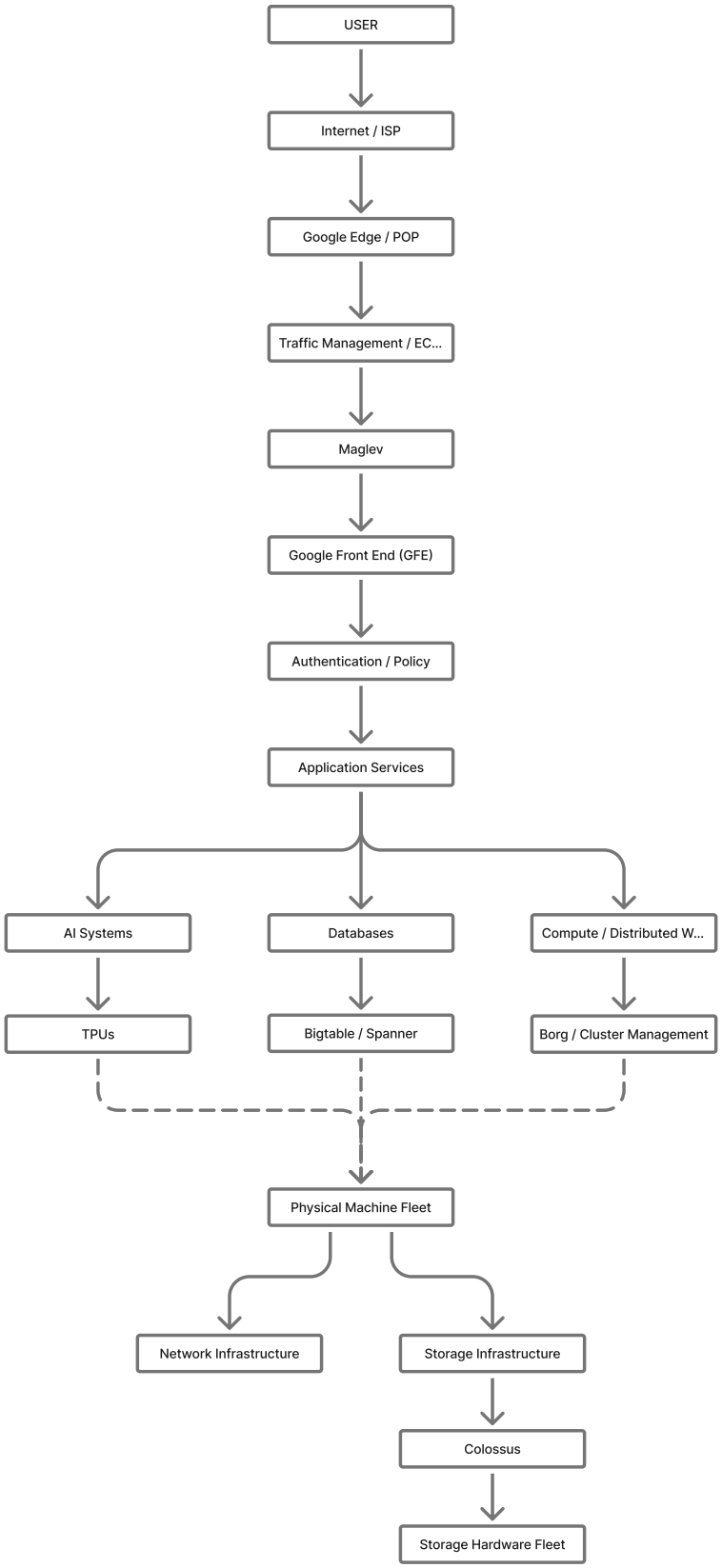
A disk is permitted to fail because the logical file is not conceptually bound to that disk.

A network path is permitted to fail because connectivity is defined by the network fabric rather than by one link.

This is the foundation for what can be described as a **software-defined computing environment**.

7 A Unified Architectural Map

A public high-level model of the infrastructure is:



This map is intentionally conceptual.

Google does not publish a complete, current packet-by-packet production blueprint for every consumer product. The actual path depends on service, geography, traffic conditions, software generation, and internal implementation.

The architectural value of the map lies in the relationships among layers.

8 The Edge and Traffic Layer

8.1 From User to Google's Network

A request first traverses the user's local network and upstream Internet infrastructure before reaching Google's edge.

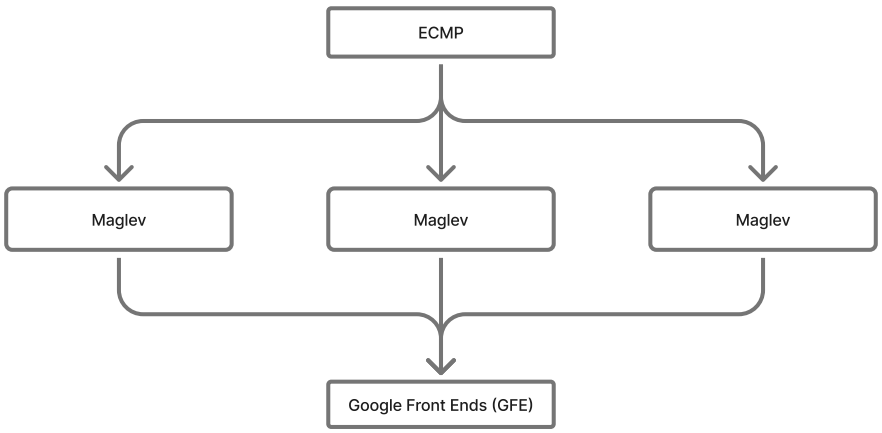
At the edge, Google can place infrastructure close to users and use distributed traffic-management systems to guide traffic into the appropriate service infrastructure.

The supplied research dossier identifies Google Global Cache (GGC) as an important part of the historical/public edge architecture for locally serving cached content. The exact role of individual edge systems varies by product and generation.

8.2 Software Load Balancing with Maglev

Maglev is Google's publicly documented software load-balancing architecture.

A simplified view is:



The architectural significance of Maglev is that load balancing becomes a **distributed software service** rather than a vertically scaled hardware appliance.

This allows the system to:

- add capacity by adding nodes;
- avoid a single load-balancer bottleneck;
- route around node failures;
- use the same fleet-level engineering philosophy employed elsewhere in Google's infrastructure.

Maglev uses consistent-distribution techniques designed to preserve backend affinity while minimizing disruption when the set of backends changes.

The source dossier also describes ECMP at the network layer, allowing traffic to be spread across multiple Maglev instances.

8.3 Google Front Ends

The Google Front End (GFE) represents another abstraction boundary.

At a high level, the front end can:

- terminate secure connections;
- enforce routing and policy;
- authenticate or authorize requests where appropriate;
- convert network traffic into service-level requests;
- forward requests to backend services.

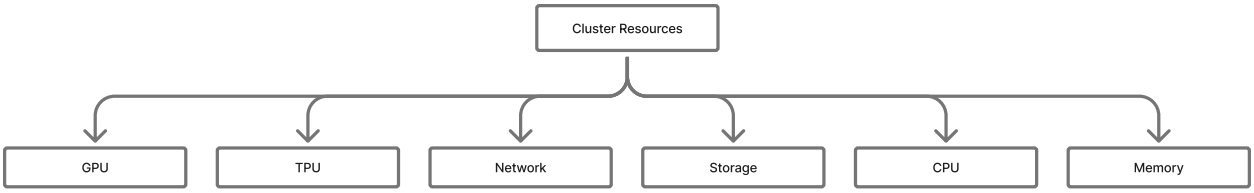
The precise relationship among GFE, authentication services, and individual product-specific systems varies across services.

9 Compute: Borg and Fleet-Level Scheduling

9.1 The Scheduling Problem

Once a request reaches an application service, it must run somewhere.

Suppose a cluster contains:



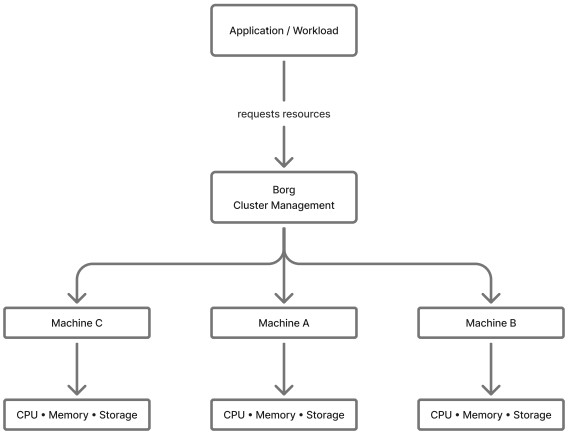
and thousands of workloads compete for them.

The developer should not need to specify a physical server. The system should reason about resource requirements, workload priority, machine health, placement constraints, and failure domains.

9.2 Borg as a Resource Abstraction

Google's Borg publication describes a large-scale cluster-management system providing scheduling, admission control, task packing, overcommitment, resource isolation, monitoring, naming, and fault recovery.

The abstraction is:



An application requests resources rather than machines.

This allows the cluster to act as a pool.

9.3 Scheduling as Resource Packing

Suppose:

```
Machine A: 16 CPU, 64 GB
Machine B:  8 CPU, 128 GB
Machine C: 32 CPU, 64 GB
```

and the workloads require different combinations of CPU and memory.

A scheduler must place them so that the resources are efficiently utilized while honoring constraints and protecting critical workloads.

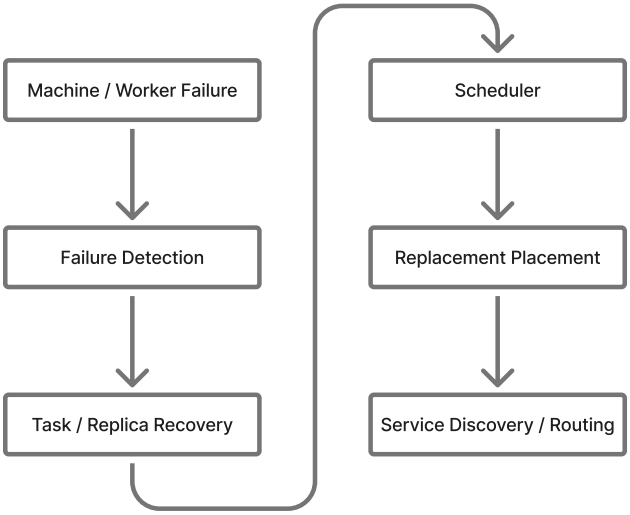
This is a multidimensional packing problem.

Google's Borg design makes machine sharing and overcommitment important because fleet utilization matters economically. Unused CPU still represents purchased silicon, power, cooling, and floor space.

9.4 Failure Recovery

A critical consequence of cluster-level scheduling is that applications do not have to remain tied to a failed machine.

The basic model is:



The exact implementation and recovery time vary by workload, but the architectural principle is general: **failure recovery is part of scheduling rather than a separate exceptional mechanism**.

9.5 Borg and Kubernetes

The supplied documents correctly emphasize Borg's influence on Kubernetes.

Kubernetes is not simply a public copy of Borg, but Borg demonstrated several ideas later reflected in container orchestration:

- declarative workload placement;
- cluster-level resource management;

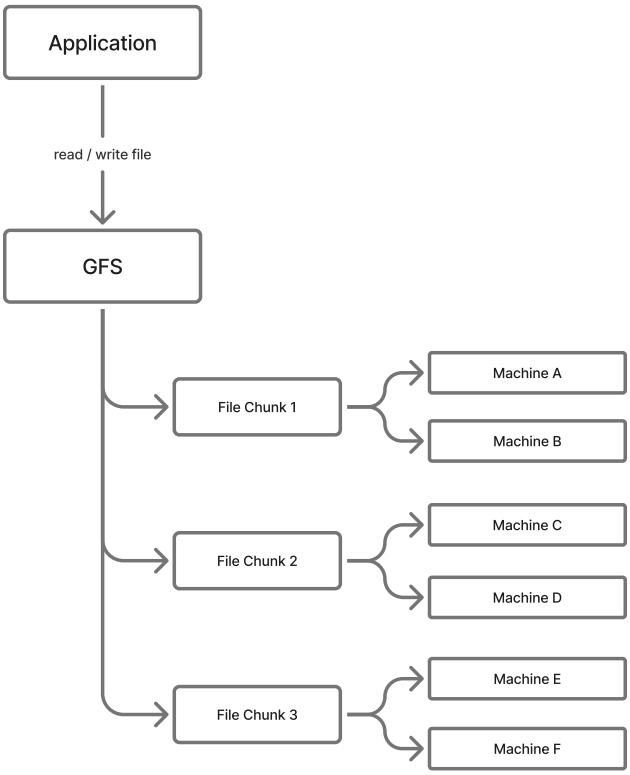
- task recovery;
- service discovery;
- fleet abstraction.

10 Storage: From GFS to Colossus

10.1 GFS

GFS was designed around large files, large-scale data processing, commodity hardware, and failure tolerance.

At a conceptual level:



The filesystem abstracts physical disk placement.

That abstraction becomes essential once data spans thousands of drives.

10.2 The Limits of a Single Metadata Master

The original GFS architecture included a centralized master responsible for metadata. This model was effective for the workload it targeted, but much larger and more diverse environments demanded more scalable metadata management.

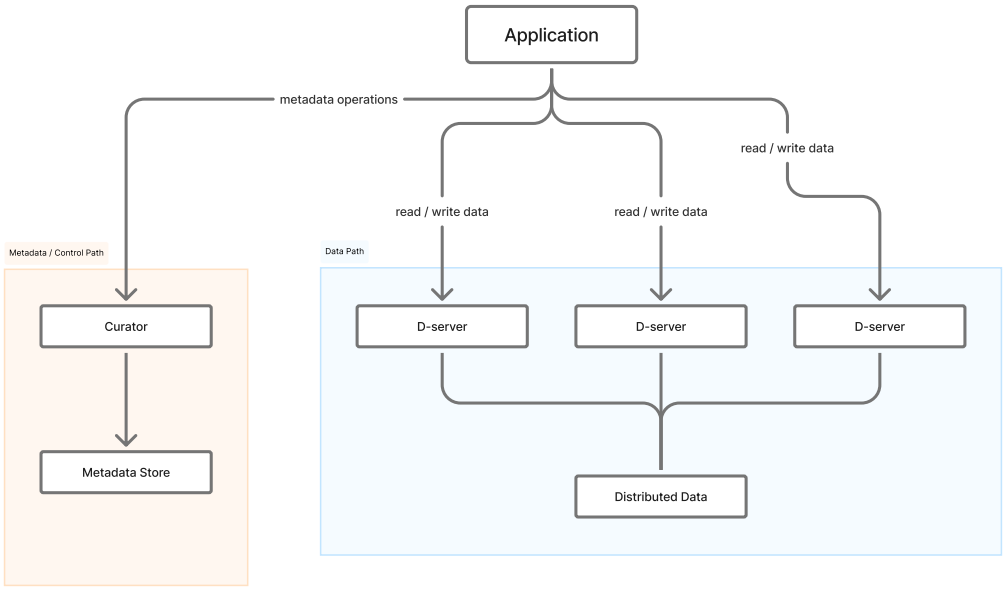
Colossus evolved the architecture.

11 Colossus: Distributed Storage at Cluster Scale

11.1 Metadata and Data Paths

The supplied papers describe Colossus as separating metadata operations from data movement.

A simplified architecture is:



The crucial idea is that the metadata authority does not sit in the middle of every byte transfer.

Instead:

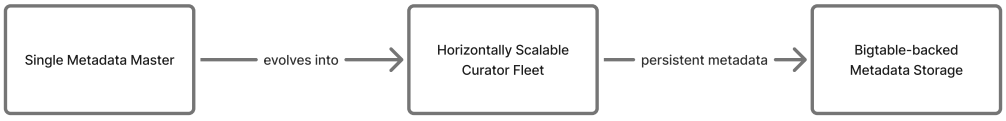
1. a client requests metadata;
2. metadata identifies relevant storage locations;
3. data can then move directly between the client and storage servers.

This avoids creating a centralized throughput bottleneck.

11.2 Curators and Bigtable-Backed Metadata

The supplied documents describe horizontally scalable **Curators** for metadata operations and Bigtable-backed metadata storage.

This is a major architectural evolution:



The change reflects the general Google strategy of removing central bottlenecks by distributing control itself.

11.3 D-Servers and Storage Devices

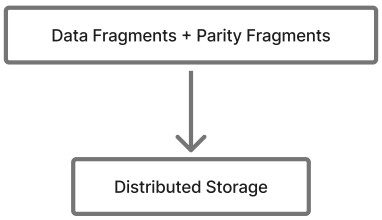
The data layer consists of storage servers holding physical devices.

From the application's point of view, these are not individual disks. The logical file is an abstraction over the distributed physical layout.

11.4 Erasure Coding

The source documents describe the use of erasure coding, including Reed–Solomon techniques, as part of the storage architecture.

The basic principle is:



rather than:

Erasure coding can reduce storage overhead relative to naive full replication, while still allowing reconstruction after failures.

The cost is increased computation and network activity during encoding, repair, and reconstruction.

11.5 Integrity and Background Repair

The research dossier describes several mechanisms that contribute to durability:

- checksums for detecting corruption;
- background scanning;
- proactive repair of degrading storage;
- parity-based reconstruction;
- asynchronous garbage collection.

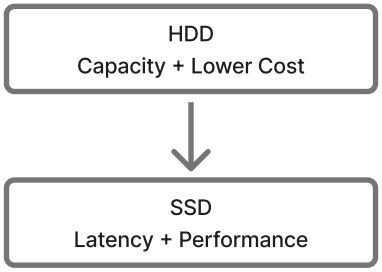
The broader principle is more important than any one implementation detail:

> **Data durability is an active process, not simply the presence of multiple copies.**

12 Storage Tiering and Modern Colossus

The updated research material describes Colossus as a storage platform that can combine high-capacity HDDs with faster SSD resources.

The conceptual trade-off is:



Workload access patterns can be used to determine which data should benefit from faster storage or caching.

The supplied research documents report public examples of:

- filesystems exceeding 10 exabytes;
- read throughput above 50 TB/s on some large filesystems;
- write throughput above 25 TB/s;
- more than 600 million IOPS in a busy cluster.

These values describe particular publicly discussed configurations and should **not** be interpreted as Google's total global storage capacity or global aggregate throughput.

The updated material also describes **Rapid Storage**, built on Colossus, as targeting AI and high-performance workloads with very high bandwidth, high request rates, and low latency. The updated draft additionally identifies **Filestore on Colossus** as an example of a cloud-facing service using the underlying storage substrate.

These developments show that storage is no longer only a capacity problem. For AI workloads, storage latency and throughput directly affect accelerator utilization.

13 Structured Data: Bigtable

A filesystem answers:

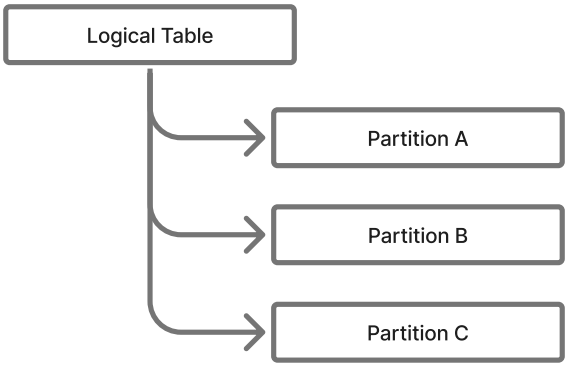
> Where are the bytes?

A structured data system answers:

> What do these bytes mean to the application?

Bigtable provides a logical table abstraction over distributed physical storage.

Its conceptual model is:



Partitions, often described as tablets in the Bigtable architecture, can be distributed across many servers.

This makes the table scalable beyond one machine.

13.1 Advantages

Partitioning provides:

- horizontal scale;
- distributed throughput;
- flexible storage expansion;
- the ability to redistribute hot or oversized partitions.

13.2 Costs

Partitioning introduces:

- cross-partition coordination;
- more complex transaction semantics;
- more complicated failure handling;
- data-movement overhead during rebalancing.

Bigtable therefore illustrates a recurring principle in Google's architecture:

> **Scaling out makes capacity easier to grow, but makes coordination harder.**

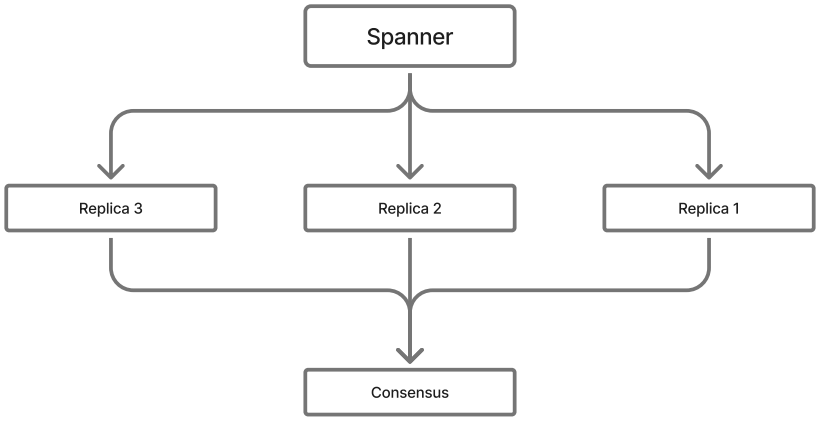
The updated source material reports that modern Bigtable operates at extremely large data volumes and request rates, demonstrating how far the original abstraction has evolved.

14 Spanner: Global State and Strong Consistency

Bigtable and similar systems can provide enormous scale, but some workloads require stronger transactional guarantees.

Spanner addresses this problem.

The public architectural model is:



Spanner combines:

- partitioning;
- synchronous replication;
- distributed transactions;
- consensus;
- geographic distribution;
- external consistency;
- an explicit time abstraction.

14.1 The Core Problem

Suppose two geographically separated transactions access related state.

Without a mechanism for global ordering, the system can observe events in ambiguous order because clocks drift and network delays vary.

Spanner therefore needs more than replication. It needs a principled method for establishing transaction order.

15 TrueTime and the Representation of Uncertainty

The supplied research documents describe **TrueTime** as an API exposing an interval rather than pretending that a distributed clock reading is perfectly precise.

Conceptually:

$$\text{TT.now}() = [\text{earliest},\ \text{latest}]$$

The uncertainty interval is bounded using timekeeping infrastructure, including GPS and atomic-clock references in the published design.

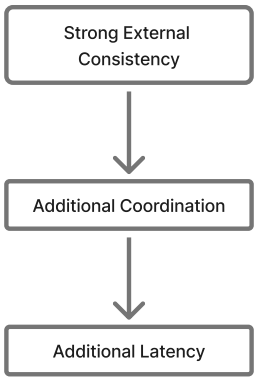
The key engineering idea is:

> **The system models clock uncertainty explicitly instead of pretending that clocks are perfectly synchronized.**

This enables Spanner to reason about transaction order.

The source dossier describes a commit-wait mechanism in which the system waits until the uncertainty interval has passed before acknowledging certain commits.

The consequence is a trade-off:

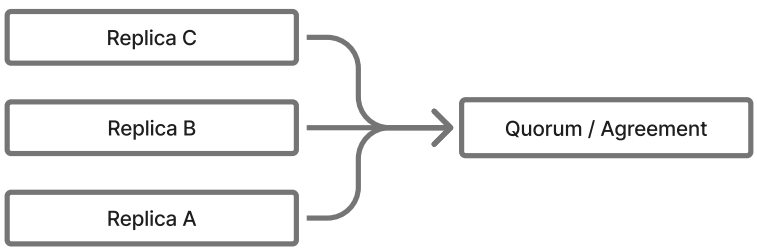


The exact magnitude of latency depends on implementation conditions and should not be generalized beyond the published evidence.

16 Consensus and Replication

Distributed replicas need agreement.

A simplified consensus process is:



A quorum provides a basis for committing state even when a minority of replicas are unavailable.

The supplied documents associate Spanner and other Google distributed systems with Paxos-based consensus.

The important abstraction is not the name of the protocol alone. It is the fact that:

> **Correctness is established through distributed agreement rather than through the assumption that every replica is always reachable.**

This creates a fundamental trade-off between coordination, latency, and availability under network partitions.

17 Networking as a Distributed Computer

Google's architecture depends on networking at every layer.

The network must connect:

- users to edge infrastructure;
- edge systems to services;
- services to databases;
- databases to storage;
- compute to compute;
- accelerators to accelerators;
- data centers to data centers.

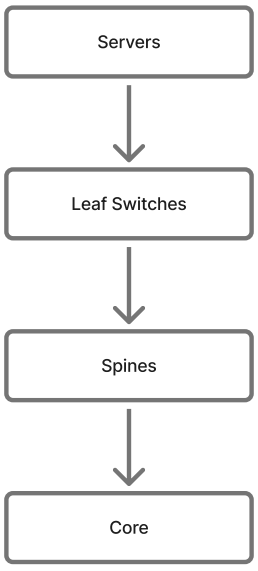
The network therefore cannot remain a passive transport mechanism.

It becomes an active computing layer.

18 Jupiter: The Data-Center Fabric

Google's publicly described Jupiter architecture uses a multi-stage Clos topology with centralized software control and high-capacity switching.

A simplified topology is:



The advantage is path diversity and aggregate bandwidth.

Rather than depending on a single network path, traffic can use many paths through the fabric.

The published Jupiter work demonstrated the ability to scale data-center networking by orders of magnitude.

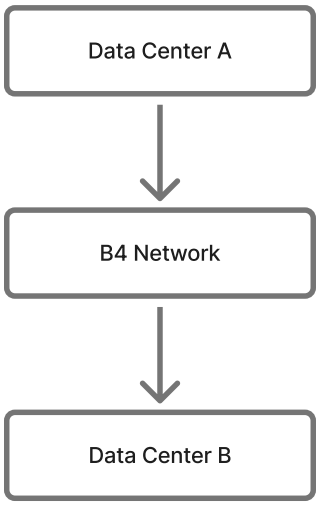
The updated research documents further associate later networking generations with optical circuit switching and other techniques designed for increasingly demanding AI and data-center traffic patterns.

19 B4: Google's Wide-Area Network

Jupiter operates primarily inside the data center.

B4 addresses the wide-area network connecting sites.

A conceptual model is:



Instead of relying entirely on distributed routing protocols to optimize all traffic locally, Google's software-defined WAN can apply a more global view of network demand and priorities.

The supplied research describes **Bandwidth Enforcer (BwE)** as a key part of traffic engineering and prioritization.

The important architectural principle is:

The operator can program the network according to application-level priorities.

For example, during congestion, interactive user-serving traffic may be prioritized relative to background transfers.

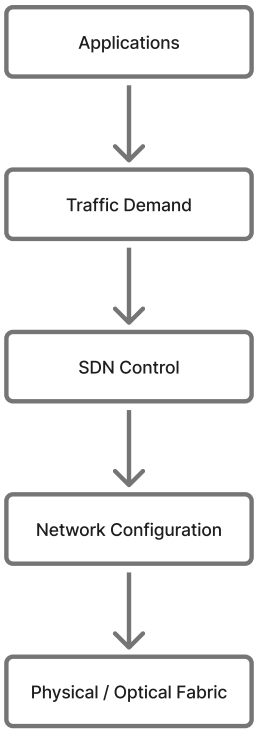
This is especially important as AI introduces enormous long-lived flows for training, parameter synchronization, data movement, and checkpointing.

20 Orion and Modern SDN Control

The supplied updated paper includes **Orion** as a distributed SDN control-plane evolution operating across Google's networking environment.

The significance of this work is that the control plane itself becomes distributed and scalable.

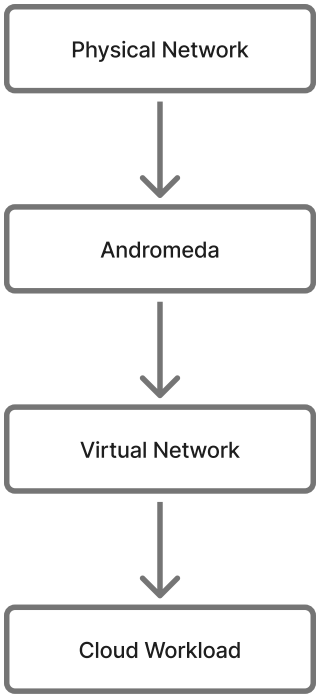
The architecture is therefore:



The network can be changed by software rather than by manual configuration of each physical device.

21 Andromeda and Cloud Network Virtualization

Google Cloud introduces another abstraction:



Andromeda provides network virtualization, isolation, provisioning, and packet-processing paths for cloud workloads.

The supplied research dossier describes a **Hoverboard** model and an OS-bypass fast path for performance-sensitive traffic.

The general engineering goal is clear:

> **Virtual networking must be fast enough that virtualization does not make the physical network unusably expensive or slow.**

The supplied source material reports extremely low packet-processing budgets for relevant datapaths. These figures should be understood as published performance characteristics of particular implementations, not as a universal latency for all cloud traffic.

22 AI Infrastructure and TPUs

The architecture is increasingly shaped by AI workloads.

General-purpose CPUs remain important because they are flexible and suitable for diverse microservices.

However, large neural-network workloads are dominated by operations that can be accelerated with specialized hardware.

Google's **Tensor Processing Units (TPUs)** are application-specific accelerators designed around those workloads.

The supplied material identifies the following characteristics for TPU v5p:

- 8,960 chips per Pod;
- approximately 459 TFLOPS of BF16 peak compute per chip;
- approximately 95 GiB of HBM per chip;
- high-bandwidth inter-chip networking;
- a 3D-torus-style interconnect.

The updated draft further describes later TPU generations, including Trillium and Ironwood, as evidence that Google continues to scale both compute and interconnect.

The most important architectural point is not the exact chip specification.

It is that the accelerator is designed as part of a **distributed system**.

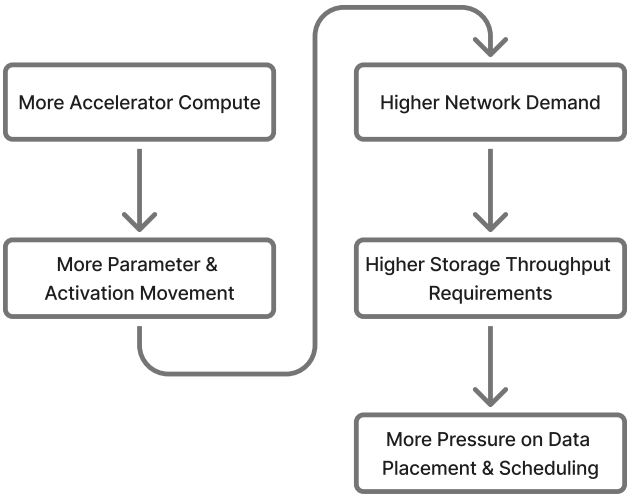
A TPU chip is not useful merely because it can execute matrix operations quickly. It must also:

- obtain model parameters;
- communicate with other accelerators;
- access training data;
- coordinate distributed computations;
- store and retrieve checkpoints;
- integrate with the scheduling and network layers.

23 The Increasing Coupling of Compute, Network, and Storage

AI changes the infrastructure map because computational throughput can become so large that data movement dominates.

A simplified chain is:



Thus, the older conceptual separation:

remains architecturally useful but becomes less absolute operationally.

The updated source material's discussion of Rapid Storage is important here. Low-latency storage can directly affect the utilization of expensive accelerators.

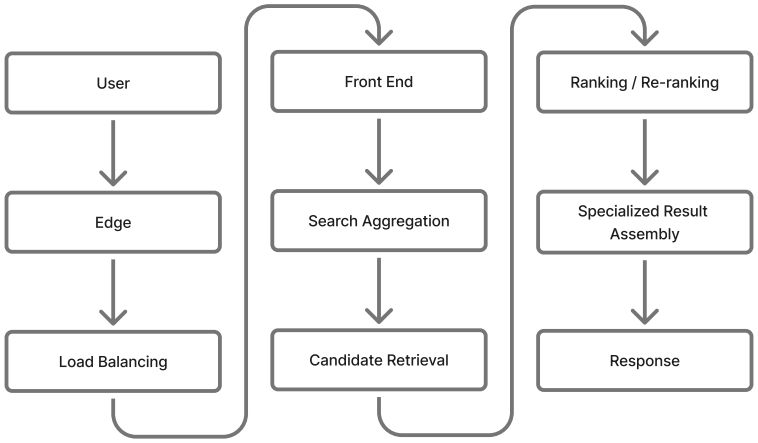
This produces a new optimization target:

> **Do not merely maximize compute throughput; maximize useful end-to-end application throughput.**

24 A Search Request as a High-Level Example

The supplied research dossier includes a more detailed Search-oriented flow involving systems such as SuperRoot, Ascorer, Twiddler, SnippetBrain, and Glue.

These terms help illustrate how a search engine can be decomposed into specialized stages:



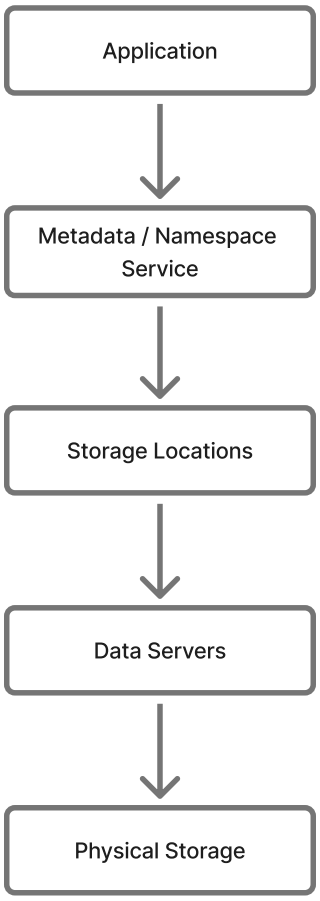
However, the exact current production Search architecture is not publicly exposed in complete form.

Therefore systems such as SuperRoot, Ascorer, and Twiddler should be interpreted as **historically or publicly discussed architectural components**, not as a definitive complete map of Google's 2026 Search stack.

The deeper lesson remains valid: complex interactive applications can decompose one request into multiple specialized distributed services.

25 Storage Write as a High-Level Example

A simplified storage operation can be represented as:



The supplied documents describe Colossus as separating metadata operations from the actual data path.

This creates an important property:

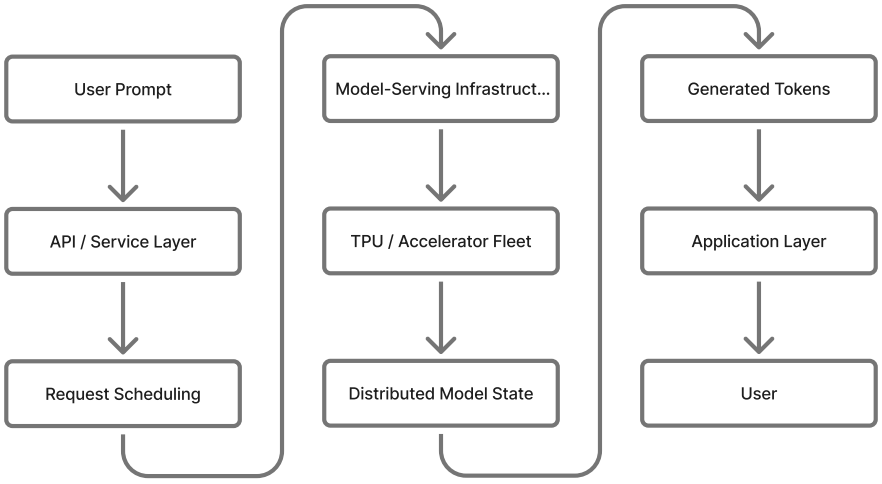
> Metadata can scale independently from bulk data movement.

The client can obtain information about where data belongs and then transfer data without sending every byte through the metadata service.

This is an example of **control-plane/data-plane separation**, a pattern that appears repeatedly throughout Google's infrastructure.

26 AI Inference as a High-Level Example

A simplified inference path is:



The exact routing, caching, batching, admission control, and model-parallel execution of production Gemini systems are proprietary.

Nevertheless, the architecture can be understood through the same general layers:

$$\text{Network} + \text{Scheduling} + \text{Compute} + \text{Interconnect} + \text{Storage}$$

This is why AI infrastructure cannot be studied as "just a GPU or TPU problem."

27 Data Architecture: Physical Storage vs. Logical State

A key distinction across the supplied research material is between **where data physically exists** and **how applications logically understand it**.

At the physical layer:

$$\text{Disk} \rightarrow \text{SSD} \rightarrow \text{Storage Server} \rightarrow \text{Filesystem}$$

At the logical layer:

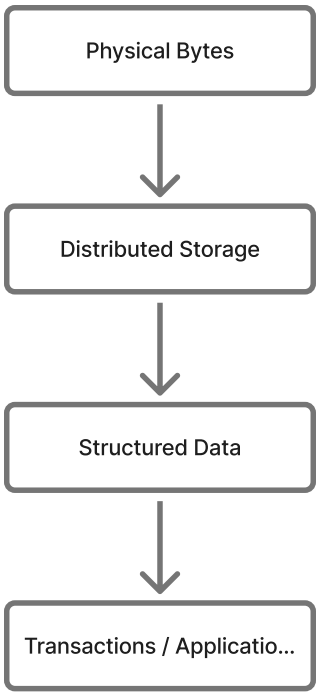
$$\text{File} \rightarrow \text{Table} \rightarrow \text{Row} \rightarrow \text{Record} \rightarrow \text{Transaction} \rightarrow \text{Object}$$

Colossus can be viewed as providing a physical storage substrate.

Bigtable can provide a structured distributed data abstraction.

Spanner can provide relational semantics and distributed transactional behavior.

Thus:



This layering is one of the most important features of Google's architecture.

28 Caching

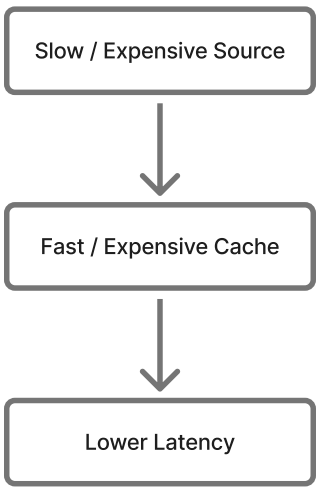
Caching appears at multiple layers.

At the network edge, caches can serve frequently requested content close to users.

Within storage, hot data can benefit from SSD placement or caching.

Within applications, services can maintain memory-resident indexes or intermediate results.

The general pattern is:



But caches introduce a consistency problem.

The system must decide:

- when cached data is valid;
- how stale it may become;
- when it should be invalidated;
- whether serving stale data is acceptable;
- how much cache capacity is worth provisioning.

Caching therefore trades correctness complexity and capacity cost for latency reduction.

29 Failure Analysis

Failure handling is cross-layer.

29.1 Compute Failure

A worker disappears.

The cluster manager detects the failure and can reschedule work on a healthy node.

The application should ideally remain associated with a logical service rather than a physical host.

29.2 Storage Failure

A disk or storage server becomes unavailable.

Redundancy or erasure coding allows the logical data to remain reconstructable.

Background systems can detect corruption, rebuild damaged data, and restore desired redundancy levels.

29.3 Network Failure

A link or network device fails.

Multi-path topologies and SDN control can reroute traffic.

This prevents a single physical path from becoming a service-wide dependency.

29.4 Database Replica Failure

A replica becomes unavailable.

Consensus-based replication can allow the system to continue if sufficient replicas remain available.

This is a key distinction:

> **The architecture tolerates specified classes of failure; it does not make failure impossible.**

29.5 Software Failure

Software failure is different from hardware failure because redundant execution of the same bad software can reproduce the same error.

The supplied research material therefore identifies techniques such as:

- staged or canary deployment;
- rollback;
- snapshots;
- controlled recovery;
- application-specific reconciliation.

The general lesson is that fault tolerance must include both hardware and software failure modes.

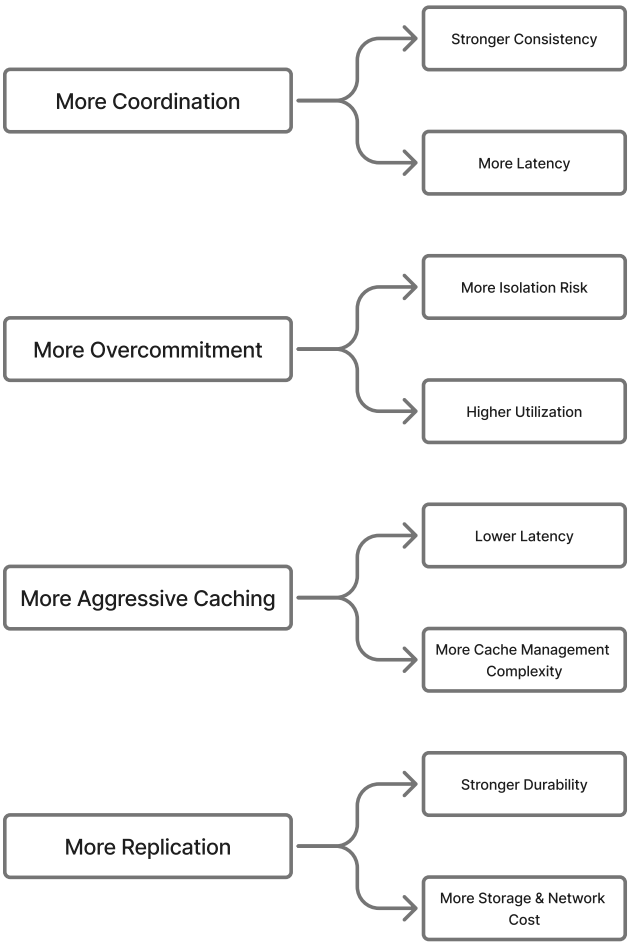
30 Performance and Tail Latency

Google's infrastructure is optimized across multiple dimensions:

- throughput;
- latency;
- tail latency;
- IOPS;
- CPU utilization;
- accelerator utilization;
- network utilization;
- power efficiency;
- storage efficiency.

These goals are not always aligned.

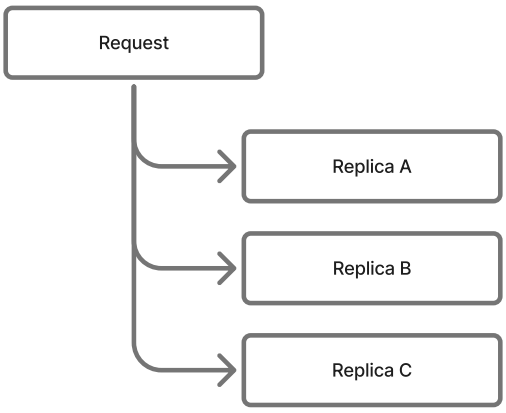
For example:



30.1 Stragglers

The supplied dossier describes **hedged requests** and aggressive straggler handling as examples of techniques for reducing the impact of slow replicas.

The general model is:



Take first acceptable result.

This costs additional work but can reduce user-visible tail latency.

Again, this illustrates Google's characteristic approach: accept some additional infrastructure cost to improve the behavior of the overall service.

31 Quantitative Scale

The source documents report several public figures that illustrate the scale of Google's infrastructure.

System	Publicly described figure	Classification
GFS	Hundreds of TB across thousands of disks and more than 1,000 machines in the original published generation	Historical
Borg	Clusters containing up to tens of thousands of machines in the published 2015 work	Historical
Jupiter	More than 1 Pbps of bisection bandwidth in the published generation	Historical
Bigtable	Approximately 10 EB in the updated source material, with very large aggregate QPS	Current public material in the supplied draft
Colossus	Some filesystems exceeding 10 EB	Current public material in the supplied draft
Colossus	More than 50 TB/s read throughput on some large filesystems	Current public material in the supplied draft
Colossus	More than 25 TB/s write throughput on some large filesystems	Current public material in the supplied draft
Colossus	More than 600 million IOPS in a busy cluster	Current/historical public material
TPU v5p	8,960 chips per Pod	Public specification cited by supplied documents

System	Publicly described figure	Classification
TPU v5p	~459 TFLOPS BF16 per chip	Public specification cited by supplied documents

These figures are not directly additive. Each refers to a different layer, system, or configuration.

The source documents also contain claims about global edge footprint, WAN growth, and fleet energy efficiency. Because these figures are highly time-sensitive and implementation-specific, they should be cited to the exact contemporary Google publication when used in a version intended for formal external publication.

32 Architectural Trade-offs

Google's architecture is not a collection of universally optimal decisions.

Each optimization has costs.

Decision	Benefit	Primary cost
Commodity hardware	Large-scale economics and flexibility	More frequent component failures
Horizontal scaling	Capacity growth	Distributed coordination
Erasure coding	Lower storage overhead than full replication	Encoding and repair cost
Strong distributed consistency	Predictable semantics	Coordination latency
Software load balancing	Elasticity and fault tolerance	Complex software datapaths
SDN	Global traffic control	Control-plane complexity
SSD caching	Lower latency	Additional cost and policy complexity
Overcommitment	Higher utilization	Isolation and scheduling complexity
Specialized AI silicon	High AI efficiency	Specialized compiler/runtime/hardware stack
Global replication	Geographic resilience	Cross-region latency and coordination

The broader principle is:

> **Google does not eliminate complexity. It moves complexity into software, automation, and fleet management.**

33 Alternatives and Why Google Built Custom Systems

The supplied research material compares Google systems with more conventional alternatives.

33.1 Storage: Enterprise SAN vs. Colossus

Traditional enterprise storage systems can provide strong reliability and centralized management, but their architecture can be constrained by vertically scaled hardware.

Google's problem was different:

$$\text{Huge Capacity} + \text{Commodity Components} + \text{Horizontal Expansion} + \text{Frequent Failure}$$

Colossus addresses that problem through software-defined distributed storage.

33.2 Database: Conventional Relational Databases vs. Spanner

A conventional relational database can provide strong transactional guarantees on one machine or a modest cluster.

Spanner exists because Google required strong transactional semantics across geographically distributed replicas at a much larger scale.

The choice is not simply:

> Spanner is better than PostgreSQL.

It is:

> Spanner was designed for a different operating point and problem domain.

33.3 Load Balancing: Hardware Appliance vs. Maglev

A hardware load balancer may be effective at enterprise scale.

At Google's traffic levels, vertically scaling the load-balancing appliance itself would become an architectural bottleneck.

Maglev replaces this with a software fleet.

33.4 AI: General-Purpose GPUs vs. Custom TPUs

GPUs provide flexibility and broad ecosystem support.

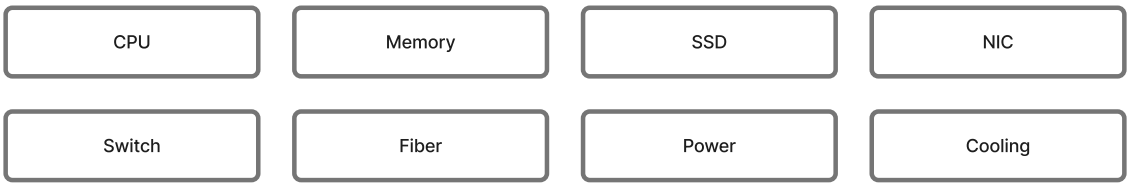
Google's TPUs provide a degree of hardware/software co-design for workloads dominated by neural-network computation.

Again, the trade-off is specialization versus generality.

34 The Importance of Abstraction

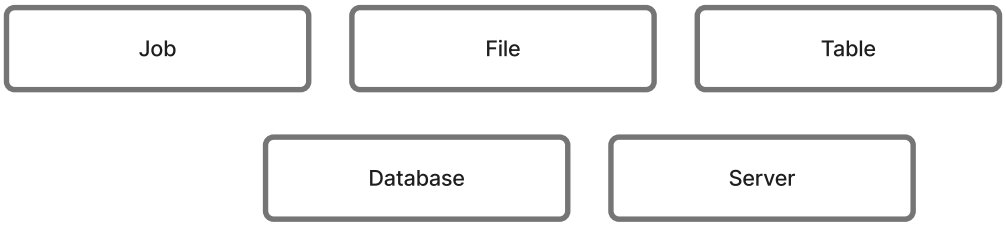
The deepest commonality across the systems is abstraction.

At the physical level:

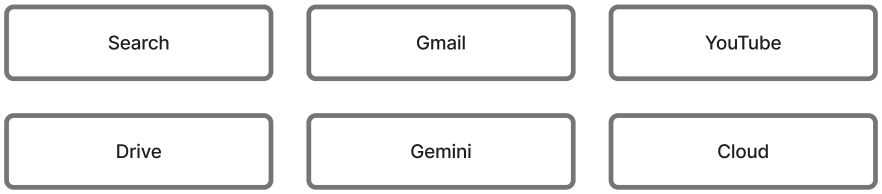


At the infrastructure level:

At the distributed-system level:



At the product level:



Applications should not need to know which physical disk contains a particular byte or which machine executes a specific task.

Instead, the infrastructure provides stable logical interfaces and manages the underlying complexity.

This is the essence of warehouse-scale computing.

35 Compute, Storage, and Networking as Semi-Independent Resources

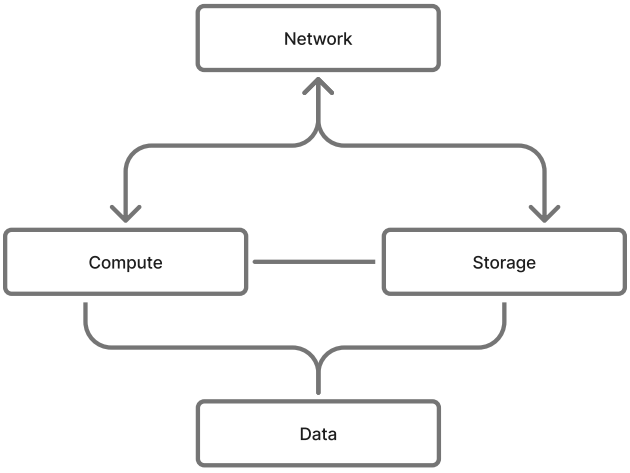
One of Google's most important design principles is **decoupling**.

If a service suddenly needs more CPU, it should not automatically require proportionally more storage.

If storage capacity grows, the system should not need to duplicate the entire compute fleet.

If traffic grows, networking should be able to scale without redesigning every application.

Thus:



The layers are independent enough to scale separately, but connected enough to cooperate efficiently.

This balance between decoupling and coordination is one of the defining characteristics of Google's infrastructure.

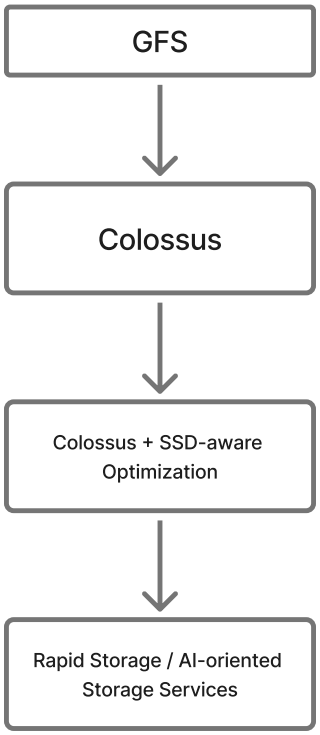
36 Current State and Architectural Evolution

The supplied updated research paper describes Google infrastructure as a continuously evolving system rather than a finished architecture.

Several trends are particularly important.

36.1 Storage

The evolution:



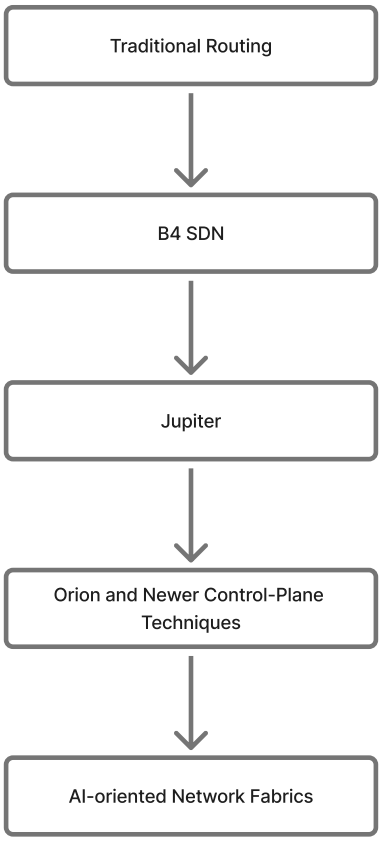
shows a move from "large durable storage" toward "large durable storage that is also fast enough for accelerator-heavy workloads."

36.2 Compute

Borg remains historically foundational, while Google's broader ecosystem continues to use cluster scheduling and orchestration ideas descended from large-scale internal infrastructure.

36.3 Networking

The evolution:



shows that networking continues to become more software-defined and more tightly integrated with workload scheduling.

36.4 AI

TPU generations continue to evolve.

The source documents emphasize that AI scale is increasingly determined by a combination of:

Accelerator Compute+HBM Capacity and Bandwidth+Interconnect Bandwidth+Storage Throughput+Scheduler Efficiency

Thus AI is not simply another application category. It is changing the balance among the infrastructure layers themselves.

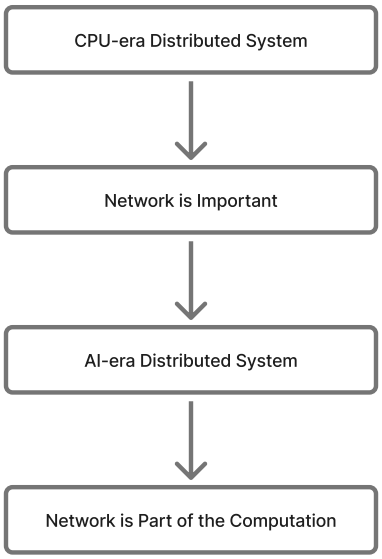
37 AI and the Changing Definition of "The Network"

Traditional data-center networking separates compute from communication.

AI training makes this separation less convenient.

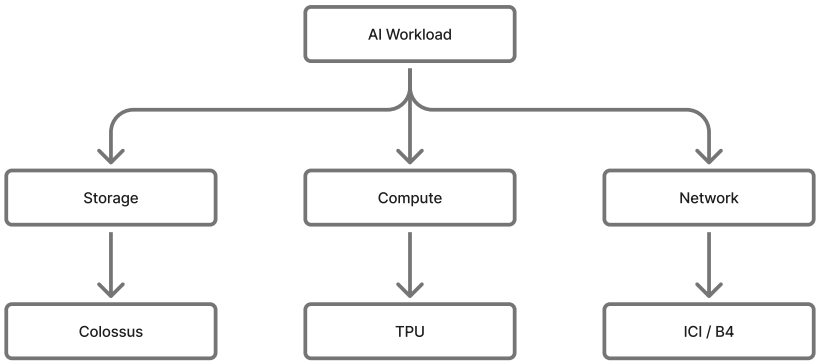
A large distributed model may continuously exchange large quantities of information among accelerators.

The consequence is:



In this environment, the interconnect can influence overall model-training efficiency almost as strongly as the accelerator itself.

The architecture therefore increasingly resembles:



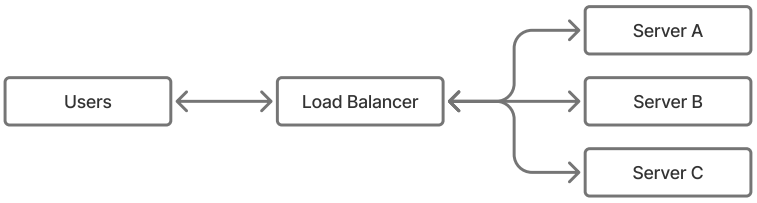
This is one of the most important architectural shifts identified across the supplied research.

38 A "Tiny Google" as a Teaching Model

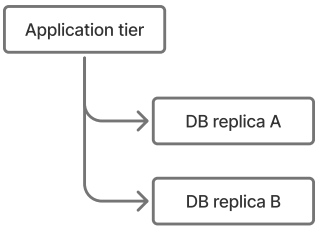
A useful way to understand the architecture is to construct it incrementally.

38.1 Step 1: One server

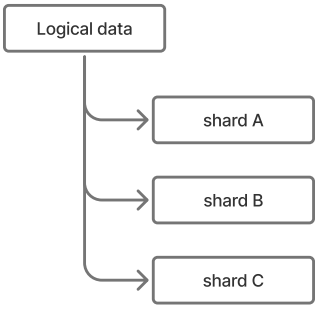
38.2 Step 2: Multiple application servers



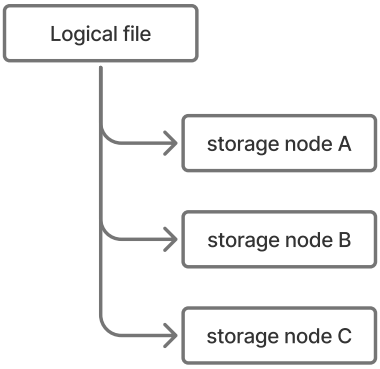
38.3 Step 3: Replicated database



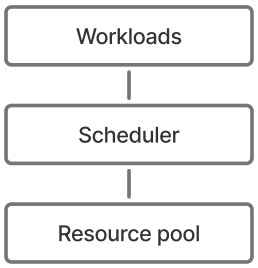
38.4 Step 4: Sharding



38.5 Step 5: Distributed storage



38.6 Step 6: Cluster scheduler



38.7 Step 7: Service discovery

Services no longer depend on static machine addresses.

38.8 Step 8: Multi-region deployment

State and services span geography.

38.9 Step 9: Consensus

Distributed replicas coordinate through an explicit agreement protocol.

38.10 Step 10: Specialized acceleration

AI workloads receive access to accelerator fleets.

At each step, the system becomes more capable but also more distributed.

This progression demonstrates a central truth:

> **Large-scale architecture is less about adding more servers than about adding the abstractions required to make more servers usable.**

39 Common Misconceptions

39.1 "Google Is One Giant Computer"

It is not a single monolithic machine.

It is a distributed computing environment composed of very large fleets of machines, storage devices, networking systems, and specialized accelerators.

39.2 "Google Searches the Live Web for Every Query"

Search generally depends on large precomputed indexes and distributed retrieval systems rather than scanning the live Internet from scratch for each request.

39.3 "Cloud Storage Is Just a Folder on a Disk"

At hyperscale, logical files and objects are abstractions over distributed storage systems.

39.4 "More Replicas Always Solve Reliability"

More copies can improve durability, but replication creates additional storage cost and does not eliminate coordination or correlated failures.

39.5 "The Network Is Just a Cable Between Servers"

At Google's scale, networking is an actively programmed distributed resource with scheduling, prioritization, load balancing, virtualization, and specialized accelerator interconnects.

40 Broader Impact

Google's infrastructure work has influenced the wider field.

40.1 Borg and Kubernetes

Borg demonstrated practical fleet-level scheduling and workload orchestration at unusually large scale. Kubernetes later brought many related principles into an open ecosystem.

40.2 GFS and Distributed Storage

GFS became a landmark reference for distributed filesystems operating over commodity hardware.

40.3 Bigtable and Distributed NoSQL

Bigtable demonstrated a scalable abstraction for structured data distributed across many machines, influencing subsequent database architectures.

40.4 Spanner and Global SQL

Spanner established a practical model for globally distributed transactional databases with strong consistency guarantees.

40.5 SDN and Datacenter Networking

B4, Jupiter, and related work helped demonstrate the viability of software-defined traffic control and custom network fabrics at very large scale.

The influence of these systems extends beyond Google because the underlying problems are common to many large distributed platforms.

41 Lessons for System Designers

Although most systems do not operate at Google's scale, several principles generalize.

41.1 Design Around Logical Resources

Applications should depend on: service, file, table, queue, workload.

rather than: machine 17, disk 42, IP address 10.1.2.7.

41.2 Assume Failure

Ask:

> What happens if this component disappears?

before assuming that it remains healthy.

41.3 Scale Horizontally Where Appropriate

When a bottleneck is fundamentally scalable, adding machines may be more robust than building an increasingly large single machine.

41.4 Make Control Planes and Data Planes Distinct

Metadata and control traffic should not unnecessarily become a bottleneck for bulk data.

This pattern appears in Colossus and in Google networking.

41.5 Measure the Tail

Mean latency can hide the behavior that users actually experience.

High-percentile latency is often the metric that reveals architectural weakness.

41.6 Move Complexity to the Right Layer

A reliable system may be complex internally while remaining simple at its interface.

That is a feature, not a contradiction.

42 Limitations of Public Knowledge

A rigorous study of Google's architecture must distinguish evidence from inference.

42.1 Exact Current Server Counts

Google does not publish a complete live inventory of its global server fleet.

Therefore exact machine-count claims should not be presented as established fact without a specific contemporary source.

42.2 Exact Data-Center Topology

Google publishes architectural concepts such as Jupiter, but not the complete physical topology of every current data center.

Any diagram in this paper is therefore conceptual.

42.3 Current Search Architecture

Google has published many systems papers and technical descriptions concerning Search, but those documents do not constitute a complete 2026 production blueprint.

Components such as SuperRoot, Ascorer, and Twiddler are useful for illustrating the decomposition of Search, but the exact current production architecture remains proprietary.

42.4 Gemini and AI Serving

Public information establishes the existence of advanced TPU infrastructure and large-scale distributed AI systems.

It does not provide a complete end-to-end production map for how every Gemini request is admitted, routed, batched, cached, scheduled, and served.

42.5 Proprietary Control Policies

Google publicly documents major networking architectures, but proprietary algorithms, operational thresholds, placement policies, traffic heuristics, and control-plane details may remain undisclosed.

42.6 Historical Papers Are Not Current Blueprints

A system paper from 2003, 2012, or 2018 documents a specific architecture or design generation.

It is evidence about Google's engineering trajectory, not automatic evidence about every current implementation detail.

This distinction is essential when discussing a continuously evolving platform.

43 Open Research Questions

The synthesized material suggests several directions for further research.

43.1 AI Traffic Patterns

How has the growth of continuous LLM inference and distributed training changed east-west traffic patterns inside Google's data centers?

43.2 WAN as a Training Fabric

As AI training increasingly spans geographically separated resources, how are WAN traffic engineering policies balancing massive model-synchronization flows against latency-sensitive interactive services?

43.3 Colossus for AI

What architectural changes were required inside Colossus and adjacent services to support increasingly latency-sensitive AI checkpointing and stateful storage?

43.4 Joint Scheduling of Compute and Data

Can future schedulers make placement decisions using both compute locality and storage/network locality as first-class constraints?

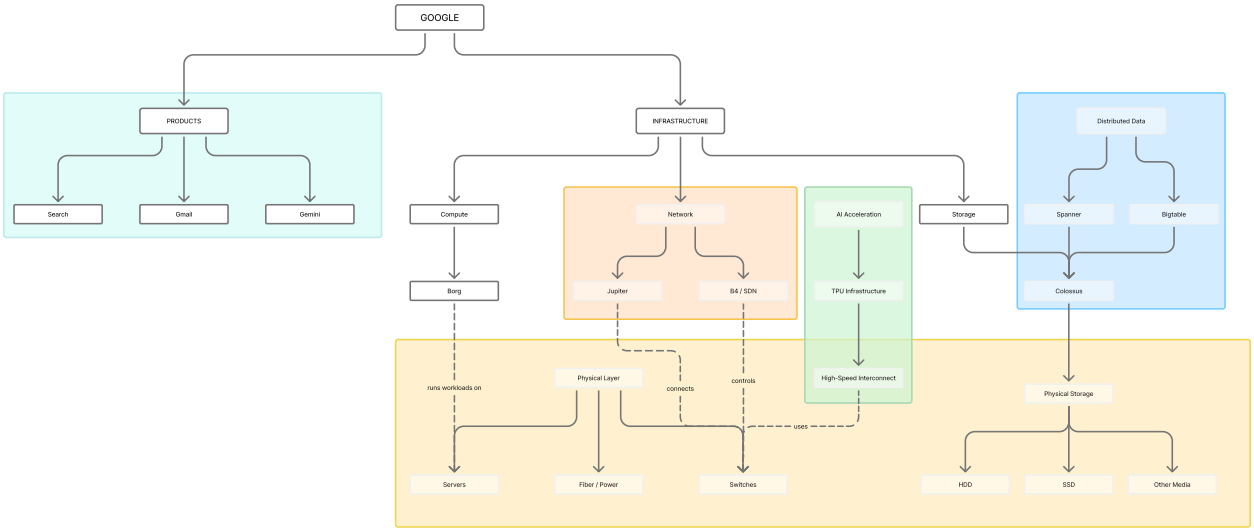
43.5 Infrastructure Energy

How will accelerator density, cooling systems, storage media, and network fabrics alter the energy profile of future warehouse-scale computers?

These questions extend naturally from the existing architecture because AI changes the relationship among resources that were historically easier to separate.

44 Final Architecture Model

The complete public mental model can be summarized as:



A user experiences a product.

The infrastructure experiences a distributed systems problem.

That distinction explains most of Google's architecture.

45 Discussion

The combined evidence points to a consistent architectural philosophy.

First, **horizontal scale replaces dependence on individual machines**. This allows capacity to grow and makes physical components replaceable.

Second, **logical abstractions replace direct physical management**. A workload requests compute; a service requests storage; a transaction requests consistency; an application consumes a network rather than a physical switch.

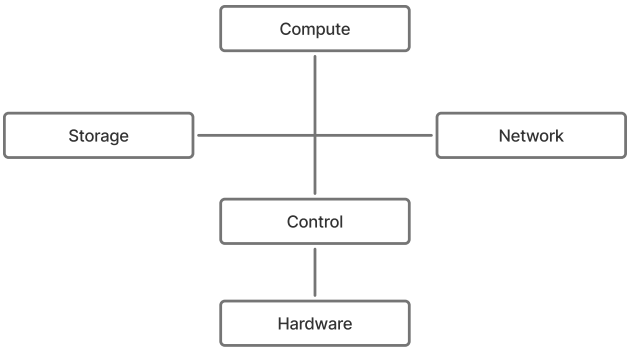
Third, **failure is incorporated into the design model**. Hardware is not assumed to be perfect. Instead, software detects failure, redirects work, rebuilds state, and preserves logical service behavior.

Fourth, **control planes become distributed systems themselves**. A storage metadata service, cluster scheduler, network controller, or consensus subsystem is not a peripheral management tool. It is part of the core infrastructure.

Fifth, **performance is end-to-end**. A faster accelerator is useless if it waits for data. A faster storage system is less useful if the network cannot deliver the data. A high-bandwidth network does not help if scheduling leaves resources idle.

Sixth, **the architecture evolves as workloads evolve**. GFS solved an earlier storage problem. Colossus addresses a newer one. CPUs once dominated the compute fleet; AI accelerators now represent another major workload class. The network once connected services; increasingly, the network participates directly in large-scale AI computation.

The resulting architecture is therefore best viewed as a set of interacting resource planes:



The control layer decides how the other layers are used.

This is why Google's infrastructure is better described as a **programmable warehouse-scale computer** than as a giant collection of servers.

46 Conclusion

Google's infrastructure is often discussed as if it were a single machine of extraordinary scale.

That is the wrong mental model.

Google is better understood as a hierarchy of distributed abstractions built over large fleets of imperfect physical resources.

At the product level: Search, Gmail, YouTube, Drive, Gemini, Cloud,

At the service level: Frontends, Application Services, Satabases, Storage, AI serving,

At the infrastructure level: Borg, Maglev, Bigtable, Spanner, Colossus, Jupiter, B4, Andromeda, TPUs,

At the physical level: CPU, memory, HDD, SSD, Network interfaces, Switches, Fiber, Power, Cooling, Data Centers.

The defining achievement is not that every component is extraordinarily powerful or perfectly reliable.

It is that the system is designed so the **logical service remains useful even when individual physical components fail, move, slow down, or disappear**.

The central thesis of this paper is therefore:

> **Google's infrastructure is a software-controlled, warehouse-scale distributed computing platform that transforms vast numbers of finite and failure-prone resources into coherent pools of compute, storage, networking, and specialized acceleration.**

The most important architectural principles are correspondingly clear:

$$\begin{aligned}&\text{Horizontal Scaling} + \text{Abstraction} + \text{Distributed Coordination}\\&+ \text{Failure Recovery} + \text{Software-Defined Networking}\\&+ \text{Decoupled Storage and Compute} + \text{Workload-Specific Acceleration}\\&= \boxed{\text{Planet-Scale Computing}}\end{aligned}$$

The architecture is not static. AI workloads are pushing the system toward tighter coupling among accelerators, interconnects, storage, scheduling, and data-center networks. Future research should therefore examine not only how Google scales individual subsystems, but how those subsystems are increasingly co-designed as one computational fabric.

The most useful conclusion is simple:

> **Google is not one giant computer. It is a system that makes many computers behave like one.**

References

Primary Google Systems Research

1. Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). **The Google File System.** *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP).*
2. Dean, J., & Ghemawat, S. (2004). **MapReduce: Simplified Data Processing on Large Clusters.** *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*
3. Chang, F., Dean, J., Ghemawat, S., et al. (2006). **Bigtable: A Distributed Storage System for Structured Data.** *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*
4. Corbett, J. C., Dean, J., Epstein, M., et al. (2012). **Spanner: Google's Globally-Distributed Database.** *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*
5. Verma, A., Pedrosa, L., Korupolu, M. R., et al. (2015). **Large-scale cluster management at Google with Borg.** *Proceedings of the 10th European Conference on Computer Systems (EuroSys).*
6. Eisenbud, D. E., Yi, C., Contavalli, C., et al. (2016). **Maglev: A Fast and Reliable Software Network Load Balancer.** *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*
7. Jain, S., Kumar, A., Mandal, S., et al. (2013). **B4: Experience with a Globally Deployed Software Defined WAN.** *Proceedings of ACM SIGCOMM.*
8. Singh, A., Ong, J., Agarwal, A., et al. (2015/2016). **Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network.** *Communications of the ACM.*
9. Dalton, M., Schultz, D., Arefin, A., et al. (2018). **Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization.** *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*
10. Vahdat, A., et al. (2021). **Orion: Google's Software-Defined Networking Control Plane.** Google Research publication.
11. Dean, J., & Barroso, L. A. (2013). **The Tail at Scale.** *Communications of the ACM*, 56.
12. Barroso, L. A., Hölzle, U., & Ranganathan, P. (2018; subsequent editions). **The Datacenter as a Computer: Designing Warehouse-Scale Machines.** Morgan & Claypool.

Google Cloud and Current Public Engineering Material Referenced in the Supplied Research

13. Hildebrand, D., & Serenyi, D. (2021). **Colossus under the hood: a peek into Google's scalable storage system.** Google Cloud.
14. Greenfield, L., & Pollen, S. (2025). **Colossus under the hood: How we deliver SSD performance at HDD prices.** Google Cloud.
15. Serenyi, D., & Saraswat, V. (2025). **Colossus: the secret ingredient in Rapid Storage's high performance.** Google Cloud.
16. Google Cloud. **TPU v5p: System Architecture and Specifications.**

17. Google Cloud. **Rapid Storage / Colossus engineering material**, as cited in the updated supplied research paper.
18. Google Cloud. **Filestore on Colossus**, as cited in the August 2026 updated supplied research paper.
19. Google Research / SIGMOD-era publication. **Twenty Years of Bigtable**, as cited in the updated supplied research paper.
20. Google Cloud. **Spanner multi-model and columnar-engine material**, as cited in the updated supplied research paper.
21. Google Cloud. **Trillium and Ironwood (TPU) public architecture and performance material**, as cited in the updated supplied research paper.

Source Corpus

22. **Deep Research Dossier: How Google Actually Works — GROUP 0: THE MAP.** User-provided research dossier.
23. **How Google Actually Works: A Technical Map of Google's Large-Scale Computing Architecture.** User-provided research-paper draft.
24. **How Google Actually Works: A Technical Map of Google's Large-Scale Computing Architecture — Updated Research Paper.** User-provided updated draft.
25. **Generic unified research-paper synthesis template.** User-provided methodological draft; used for structural guidance rather than as factual evidence.

Source and Evidence Notes

This paper intentionally distinguishes between different evidence classes.

Historically documented: claims grounded in published Google systems papers describing a particular architecture or generation.

Current public: claims derived from later public Google Cloud or Google Research material included in the supplied updated research.

Architectural synthesis: high-level request flows and relationships assembled from multiple public descriptions.

Proprietary / unknown: implementation details that cannot be established from the supplied material or public documentation.

Where the supplied research documents contained highly specific implementation claims that were identified within the source corpus itself as uncertain, inferred, or proprietary—for example, exact current Search ranking internals or complete Gemini routing—the final paper has retained them only as qualified architectural examples rather than as verified descriptions of Google's current production system.

Final Thesis

> **The fundamental architecture of Google is not a giant machine but a giant abstraction: a software-controlled distributed system that makes enormous populations of machines, disks, network links, and accelerators usable as if they were a coherent computing platform.**