CHAPTER 2

Workloads and Software Infrastructure

2.1 WAREHOUSE DATA CENTER SYSTEMS STACK

The applications that run on warehouse-scale computers (WSCs) dominate many system design trade-off decisions. This chapter outlines some of the distinguishing characteristics of software that runs in large internet services and the system software and tools needed for a complete computing platform. Here are some terms used to describe the different software layers in a typical WSC deployment.

- *Platform-level software:* The common firmware, kernel, operating system distribution, and libraries expected to be present in all individual servers to abstract the hardware of a single machine and provide a basic machine abstraction layer.

- *Cluster-level infrastructure:* The collection of distributed systems software that manages resources and provides services at the cluster level. Ultimately, we consider these services as an operating system for a data center. Examples are distributed file systems, schedulers and remote procedure call (RPC) libraries, as well as programming models that simplify the usage of resources at the scale of data centers, such as MapReduce [DG08], Dryad [Isa+07], Hadoop [Hadoo], Sawzall [Pik+05], BigTable [Cha+06], Dynamo [DeC+07], Dremel [Mel+10], Spanner [Cor+12], and Chubby [Bur06].

- *Application-level software:* Software that implements a specific service. It is often useful to further divide application-level software into online services and offline computations, since they tend to have different requirements. Examples of online services are Google Search, Gmail, and Google Maps. Offline computations are typically used in large-scale data analysis or as part of the pipeline that generates the data used in online services, for example, building an index of the web or processing satellite images to create map tiles for the online service.

- *Monitoring and development software:* Software that keeps track of system health and availability by monitoring application performance, identifying system bottlenecks, and measuring cluster health.

L. A. Barroso et al., *The Datacenter as a Computer*

Figure 2.1 summarizes these layers as part of the overall software stack in WSCs.

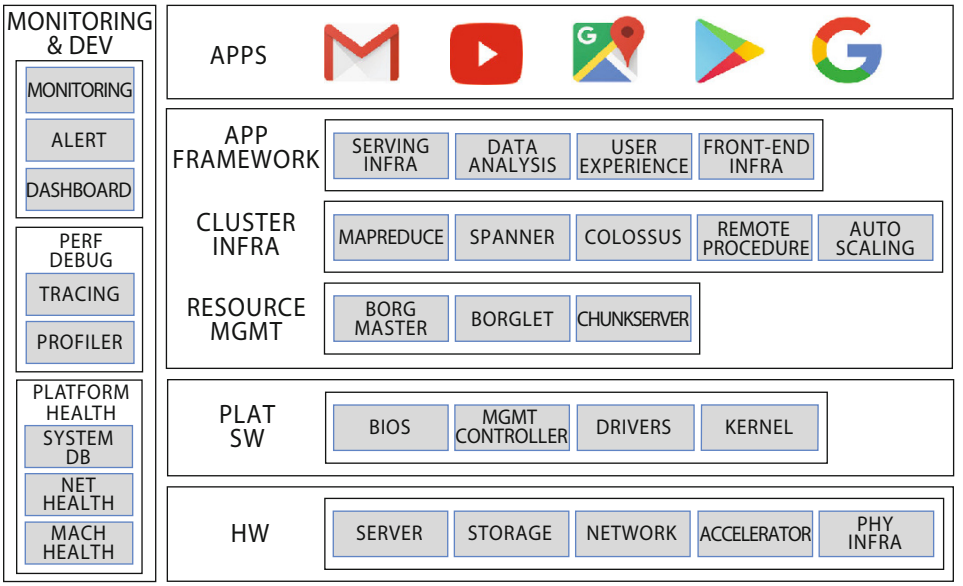

Figure 2.1: Overview of the Google software stack in warehouse-scale computers.

2.2 PLATFORM-LEVEL SOFTWARE

The basic software system image running in WSC server nodes isn't much different than what one would expect on a regular enterprise server platform. Therefore we won't go into detail on this level of the software stack.

Firmware, device drivers, or operating system modules in WSC servers can be simplified to a larger degree than in a general purpose enterprise server. Given the higher degree of homogeneity in the hardware configurations of WSC servers, we can streamline firmware and device driver development and testing since fewer combinations of devices will exist. In addition, a WSC server is deployed in a relatively well known environment, leading to possible optimizations for increased performance. For example, the majority of the networking connections from a WSC server will be to other machines within the same building, and incur lower packet losses than in long-distance internet connections. Thus, we can tune transport or messaging parameters (timeouts, window sizes, and so on) for higher communication efficiency.

Virtualization first became popular for server consolidation in enterprises but now is also popular in WSCs, especially for Infrastructure-as-a-Service (IaaS) cloud offerings [VMware]. A virtual machine provides a concise and portable interface to manage both the security and performance isolation of a customer's application, and allows multiple guest operating systems to

co-exist with limited additional complexity. The downside of VMs has always been performance, particularly for I/O-intensive workloads. In many cases today, those overheads are improving and the benefits of the VM model outweigh their costs. The simplicity of VM encapsulation also makes it easier to implement live migration (where a VM is moved to another server without needing to bring down the VM instance). This then allows the hardware or software infrastructure to be upgraded or repaired without impacting a user's computation. Containers are an alternate popular abstraction that allow for isolation across multiple workloads on a single OS instance. Because each container shares the host OS kernel and associated binaries and libraries, they are more lightweight compared to VMs, smaller in size and much faster to start.

2.3 CLUSTER-LEVEL INFRASTRUCTURE SOFTWARE

Much like an operating system layer is needed to manage resources and provide basic services in a single computer, a system composed of thousands of computers, networking, and storage also requires a layer of software that provides analogous functionality at a larger scale. We call this layer the *cluster-level infrastructure*. Three broad groups of infrastructure software make up this layer.

2.3.1 RESOURCE MANAGEMENT

This is perhaps the most indispensable component of the cluster-level infrastructure layer. It controls the mapping of user tasks to hardware resources, enforces priorities and quotas, and provides basic task management services. In its simplest form, it is an interface to manually (and statically) allocate groups of machines to a given user or job. A more useful version would present a higher level of abstraction, automate allocation of resources, and allow resource sharing at a finer level of granularity. Users of such systems would be able to specify their job requirements at a relatively high level (for example, how much CPU performance, memory capacity, and networking bandwidth) and have the scheduler translate those requirements into an appropriate allocation of resources.

Kubernetes (www.kubernetes.io) is a popular open-source program which fills this role that orchestrates these functions for container-based workloads. Based on the ideas behind Google's cluster management system, Borg, Kubernetes provides a family of APIs and controllers that allow users to specify tasks in the popular Open Containers Initiative format (which derives from Docker containers). Several patterns of workloads are offered, from horizontally scaled stateless applications to critical stateful applications like databases. Users define their workloads' resource needs and Kubernetes finds the best machines on which to run them.

It is increasingly important that cluster schedulers also consider power limitations and energy usage optimization when making scheduling decisions, not only to deal with emergencies (such as cooling equipment failures) but also to maximize the usage of the provisioned data center power budget. Chapter 5 provides more detail on this topic, and more information can be found

in recent publications [FF05, Lim+13]. Similarly, cluster scheduling must also consider correlated failure domains and fault tolerance when making scheduling decisions. This is discussed further in Chapter 7.

2.3.2 CLUSTER INFRASTRUCTURE

Nearly every large-scale distributed application needs a small set of basic functionalities. Examples are reliable distributed storage, remote procedure calls (RPCs), message passing, and cluster-level synchronization. Implementing this type of functionality correctly with high performance and high availability is complex in large clusters. It is wise to avoid re-implementing such tricky code for each application and instead create modules or services that can be reused. Colossus (successor to GFS) [GGL03, Ser17], Dynamo [DeC+07], and Chubby [Bur06] are examples of reliable storage and lock services developed at Google and Amazon for large clusters.

Many tasks that are amenable to manual processes in a small deployment require a significant amount of infrastructure for efficient operations in large-scale systems. Examples are software image distribution and configuration management, monitoring service performance and quality, and triaging alarms for operators in emergency situations. The Autopilot system from Microsoft [Isa07] offers an example design for some of this functionality for Windows Live data centers. Monitoring the overall health of the hardware fleet also requires careful monitoring, automated diagnostics, and automation of the repairs workflow. Google's System Health Infrastructure, described by Pinheiro et al. [PWB07], is an example of the software infrastructure needed for efficient health management. Finally, performance debugging and optimization in systems of this scale need specialized solutions as well. The X-Trace [Fon+07] system developed at UC Berkeley is an example of monitoring infrastructure aimed at performance debugging of large distributed systems.

2.3.3 APPLICATION FRAMEWORK

The entire infrastructure described in the preceding paragraphs simplifies the deployment and efficient usage of hardware resources, but it does not fundamentally hide the inherent complexity of a large scale system as a target for the average programmer. From a programmer's standpoint, hardware clusters have a deep and complex memory/storage hierarchy, heterogeneous components, failure-prone components, varying adversarial load from other programs in the same system, and resource scarcity (such as DRAM and data center-level networking bandwidth). Some types of higher-level operations or subsets of problems are common enough in large-scale services that it pays off to build targeted programming frameworks that simplify the development of new products. Flume [Cha+10], MapReduce [DG08], Spanner [Cor+12], BigTable [Cha+06], and Dynamo [DeC+07] are good examples of pieces of infrastructure software that greatly improve programmer productivity by automatically handling data partitioning, distribution, and fault tolerance within

their respective domains. Equivalents of such software for the cloud, such as Google Kubernetes Engine (GKE), CloudSQL, AppEngine, etc. will be discussed in the discussion about cloud later in this section.

2.4 APPLICATION-LEVEL SOFTWARE

2.4.1 WORKLOAD DIVERSITY

Web search was one of the first large-scale internet services to gain widespread popularity, as the amount of web content exploded in the mid-1990s, and organizing this massive amount of information went beyond what could be accomplished with available human-managed directory services. However, as networking connectivity to homes and businesses continues to improve, offering new services over the internet, sometimes replacing computing capabilities that traditionally lived in the client, has become more attractive. Web-based maps and email services are early examples of these trends.

This increase in the breadth of services offered has resulted in a corresponding diversity in application-level requirements. For example, a search workload may not require an infrastructure capable of high-performance atomic updates and is inherently forgiving of hardware failures (because absolute precision every time is less critical in web search). This is not true for an application that tracks user clicks on sponsored links (ads). Clicks on ads are small financial transactions, which need many of the guarantees expected from a transactional database management system. Figure 2.2 presents the cumulative distribution of cycles across workloads in Google data centers. The top 50 workloads account for only about 60% of the total WSC cycles, with a long tail accounting for the rest of the cycles [Kan+15].

Once we consider the diverse requirements of multiple services, the data center clearly must be a general-purpose computing system. Although specialized hardware solutions might be a good fit for individual parts of services (we discuss accelerators in Chapter 3), the breadth of requirements makes it important to focus on general-purpose system design. Another factor against hardware specialization is the speed of workload churn; product requirements evolve rapidly, and smart programmers will learn from experience and rewrite the baseline algorithms and data structures much more rapidly than hardware itself can evolve. Therefore, there is substantial risk that by the time a specialized hardware solution is implemented, it is no longer a good fit even for the problem area for which it was designed, unless these areas are ones where there is a significant focus on hardware-software codesign. Having said that, there are cases where specialization has yielded significant gains, and we will discuss these further later.

Below we describe workloads used for web search, video serving, machine learning, and citation-based similarity computation. Our objective here is not to describe internet service workloads

in detail, especially because the dynamic nature of this market will make those obsolete by publishing time. However, it is useful to describe at a high level a few workloads that exemplify some important characteristics and the distinctions between key categories of online services and batch (offline) processing systems.

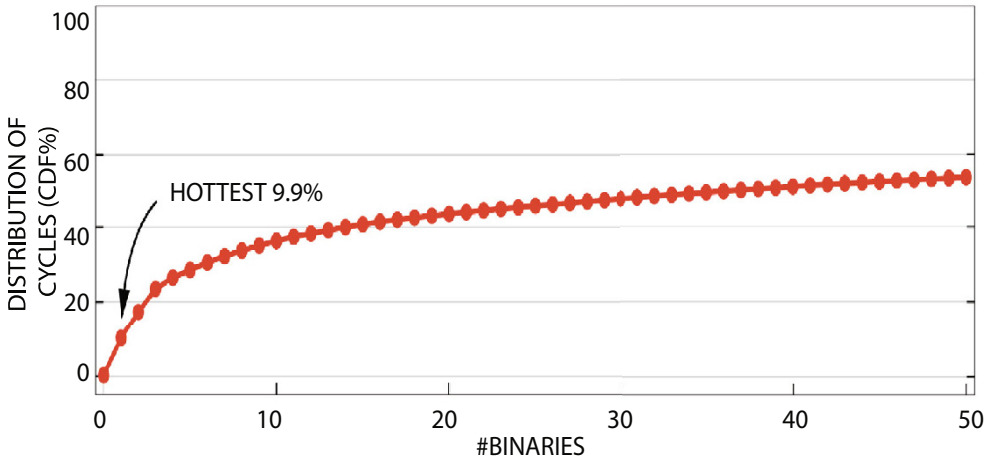

Figure 2.2: Diversity of workloads in WSCs.

2.4.2 WEB SEARCH

This is the quintessential "needle in a haystack" problem. Although it is hard to accurately determine the size of the web at any point in time, it is safe to say that it consists of trillions of individual documents and that it continues to grow. If we assume the web to contain 100 billion documents, with an average document size of 4 KB (after compression), the haystack is about 400 TB. The database for web search is an index built from that repository by inverting that set of documents to create a repository in the logical format shown in Figure 2.3.

A lexicon structure associates an ID to every term in the repository. The *termID* identifies a list of documents in which the term occurs, called a *posting list*, and some contextual information about it, such as position and various other attributes (for example, whether the term is in the document title).

The size of the resulting inverted index depends on the specific implementation, but it tends to be on the same order of magnitude as the original repository. The typical search query consists of a sequence of terms, and the system's task is to find the documents that contain all of the terms (an AND query) and decide which of those documents are most likely to satisfy the user. Queries can optionally contain special operators to indicate alternation (OR operators) or to restrict the search

to occurrences of the terms in a particular sequence (phrase operators). For brevity we focus on the more common AND query in the example below.

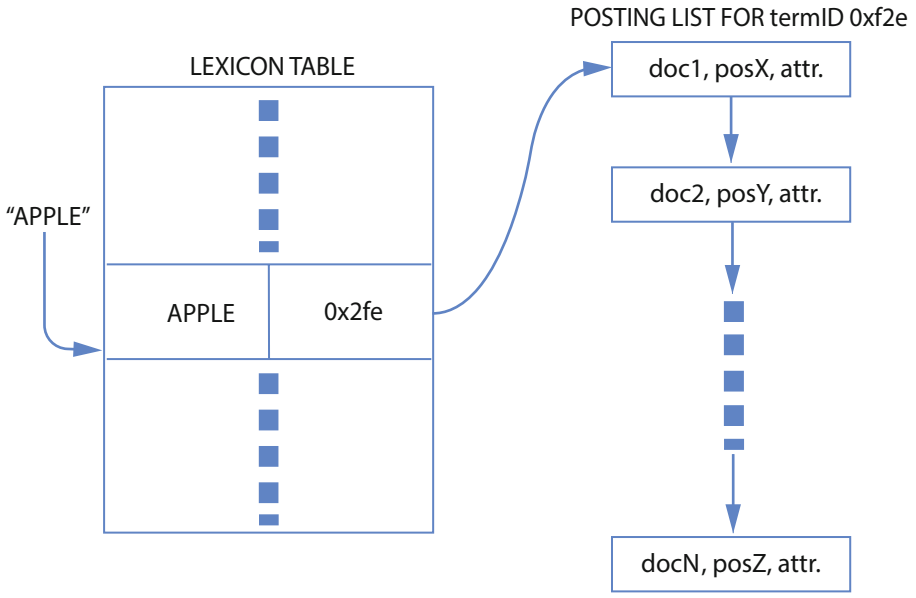

Figure 2.3: Logical view of a web index.

Consider a query such as [*new york restaurants*]. The search algorithm must traverse the posting lists for each term (*new*, *york*, *restaurants*) until it finds all documents contained in all three posting lists. At that point it ranks the documents found using a variety of parameters, such as the overall importance of the document (in Google's case, that would be the PageRank score [PB] as well as other properties such as number of occurrences of the terms in the document, positions, and so on) and returns the highest ranked documents to the user.

Given the massive size of the index, this search algorithm may need to run across a few thousand machines. That is accomplished by splitting (or *sharding*) the index into load-balanced subfiles and distributing them across all of the machines. Index partitioning can be done by document or by term. The user query is received by a front-end web server and distributed to all of the machines in the index cluster. As necessary for throughput or fault tolerance, multiple copies of index subfiles can be placed in different machines, in which case only a subset of the machines is involved in a given query. Index-serving machines compute local results, pre-rank them, and send their best results to the front-end system (or some intermediate server), which selects the best results from across the whole cluster. At this point, only the list of doc_IDs corresponding to the resulting web page hits is known. A second phase is needed to compute the actual title, URLs, and a query-specific document snippet that gives the user some context around the search terms. This

phase is implemented by sending the list of doc_IDs to a set of machines containing copies of the documents themselves. Once again, a repository this size needs to be partitioned (sharded) and placed in a large number of servers.

The total user-perceived latency for the operations described above needs to be a fraction of a second; therefore, this architecture places heavy emphasis on latency reduction. However, high throughput is also a key performance metric because a popular service may need to support many thousands of queries per second. The index is updated frequently, but in the time granularity of handling a single query, it can be considered a read-only structure. Also, because there is no need for index lookups in different machines to communicate with each other except for the final merge step, the computation is very efficiently parallelized. Finally, further parallelism is available by exploiting the fact that there are no logical interactions across different web search queries.

If the index is sharded by doc_ID, this workload has relatively small networking requirements in terms of average bandwidth because the amount of data exchanged between machines is typically not much larger than the size of the queries themselves (about a hundred bytes or so) but does exhibit some bursty behavior. Basically, the servers at the front-end act as traffic amplifiers as they distribute a single query to a very large number of servers. This creates a burst of traffic not only in the request path but possibly also on the response path as well. Therefore, even if overall network utilization is low, careful management of network flows is needed to minimize congestion.

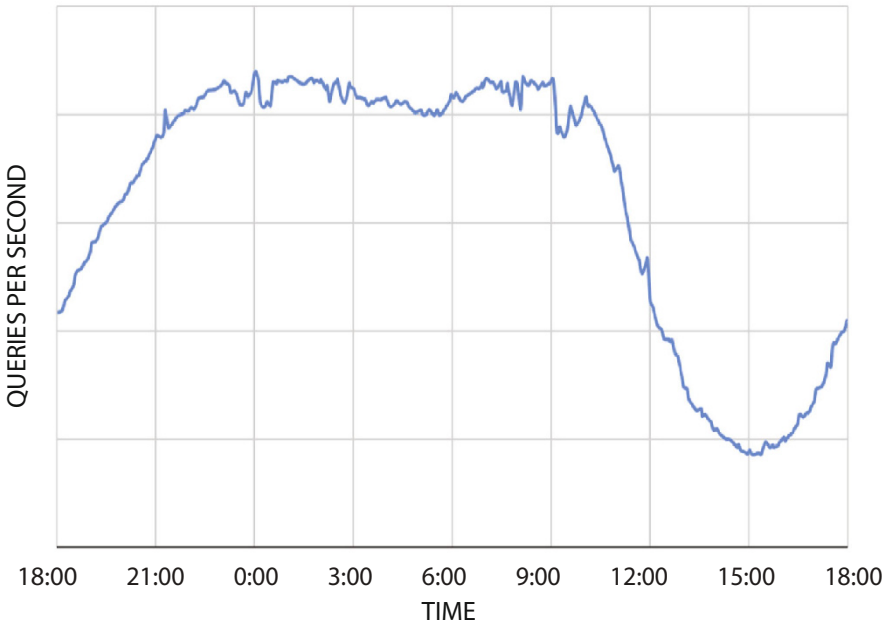

Figure 2.4: Example of daily traffic fluctuation for a search service in one data center over a 24-hr period.

Finally, because web search is an online service, it suffers from normal traffic variations because users are more active on the web at different times of the day. Figure 2.4 illustrates this effect, showing that traffic at peak usage hours can be more than twice as high as off-peak periods. Such variability presents a challenge to system operators because the service must be sized for traffic intensities significantly higher than average.

2.4.3 VIDEO SERVING

IP video traffic represented 73% of the global internet in 2016, and is expected to grow to 83% by 2021. Live video will grow 15-fold, and video-on-demand will grow double by then. In July 2015, users were uploading 400 hr of video per minute to YouTube, and in February 2017 users were watching 1 billion hours of YouTube video per day. Video transcoding (decoding video in one format and encoding it into a different format) is a crucial part of any video sharing infrastructure: video is uploaded in a plethora of combinations of format, codec, resolution, frame rate, color space, and so on. These videos need to be converted to the subset of codecs, resolutions, and formats that client devices can play, and adapted to the network bandwidth available to optimize the user experience.

Video serving has three major cost components: the compute costs due to video transcoding, the storage costs for the video catalog (both originals and transcoded versions), and the network egress costs for sending transcoded videos to end users. Improvements in video compression codecs improve the storage and egress costs at the expense of higher compute costs. The YouTube video processing pipeline balances the three factors based on the popularity profile of videos, only investing additional effort on compressing highly popular videos. Below we describe how video on demand works; other video serving use cases, like on-the-fly transcoding or live video streaming, are similar at a high level but differ in the objective functions being optimized, leading to different architectures.

Every video uploaded to YouTube is first transcoded [Lot+18] from its original upload format to a temporary high-quality common intermediate format, to enable uniform processing in the rest of the pipeline. Then, the video is chunked into segments and transcoded into multiple output resolutions and codecs so that when a user requests the video, the available network bandwidth and the capabilities of the player device are matched with the best version of the video chunk to be streamed. Video chunks are distributed across multiple machines, parallelizing transcoding of each video segment into multiple formats, and optimizing for both throughput and latency. Finally, if a video is identified as highly popular, it will undergo a second video transcoding pass, where additional compute effort is invested to generate a smaller video at the same perceptual quality. This enables users to get a higher resolution version of the video at the same network bandwidth, and the higher compute costs are amortized across egress savings on many playbacks.

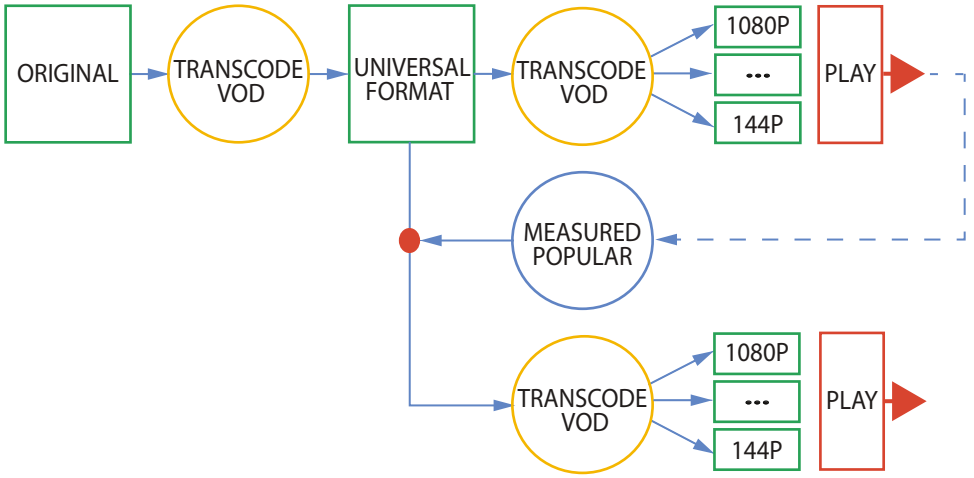

Figure 2.5: The YouTube video processing pipeline. Videos are transcoded multiple times depending on their popularity. VOD = video on demand.

Once video chunks are transcoded to their playback ready formats in the data center, they are distributed through the Edge network, which caches the most recently watched videos to minimize latency and amplify egress bandwidth. When a user requests a video that is not readily available in the Edge caches, the request is forwarded to the nearest data center or a peer Edge location, and the requested video is uploaded from the catalog.

2.4.4 SCHOLARLY ARTICLE SIMILARITY

Services that respond to user requests provide many examples of large-scale computations required for the operation of internet services. These computations are typically data-parallel workloads needed to prepare or package the data that is subsequently used by the online services. For example, computing PageRank or creating inverted index files from a web repository fall in this category. But in this section, we use a different example: finding similar articles in a repository of academic papers and journals. This is a useful feature for internet services that provide access to scientific publications, such as Google Scholar (http://scholar.google.com). Article similarity relationships complement keyword-based search systems as another way to find relevant information; after finding an article of interest, a user can ask the service to display other articles that are strongly related to the original article.

There are several ways to compute similarity scores, and it is often appropriate to use multiple methods and combine the results. With academic articles, various forms of citation analysis are known to provide good quality similarity scores. Here we consider one such type of analysis, called co-citation. The underlying idea is to count every article that cites articles A and B as a vote

for the similarity between A and B. After that is done for all articles and appropriately normalized, we obtain a numerical score for the (co-citation) similarity between all pairs of articles and create a data structure that for each article returns an ordered list (by co-citation score) of similar articles. This data structure is periodically updated, and each update then becomes part of the serving state for the online service.

The computation starts with a citation graph that creates a mapping from each article identifier to a set of articles cited by it. The input data is divided into hundreds of files of approximately the same size (for example, by taking a fingerprint of the article identifier, dividing it by the number of input files, and using the remainder as the file ID) to enable efficient parallel execution. We use a sequence of MapReduce runs to take a citation graph and produce a co-citation similarity score vector for all articles. In the first Map phase, we take each citation list (A1, A2, A3, . . . , A*n*) and generate all possible pairs of documents, and then feed them to the Reduce phase, which counts all occurrences of each pair. This first step results in a structure that associates all pairs of co-cited documents with a co-citation count. Note that this becomes much less than a quadratic explosion because most documents have a co-citation count of zero and are therefore omitted. A second MapReduce pass groups all entries for a given document, normalizes their scores, and generates a list of documents with decreasing similarity scores to the original one.

This two-pass data-parallel program executes on hundreds of servers with relatively lightweight computation in each stage followed by significant all-to-all communication between the Map and Reduce workers in each phase. Unlike web search, however, the networking traffic is streaming in nature, which makes it friendlier to existing congestion control algorithms. Also contrary to web search, the latency of individual tasks is much less important than the overall parallel efficiency of the workload.

2.4.5 MACHINE LEARNING

Deep neural networks (DNNs) have led to breakthroughs, such as cutting the error rate in an image recognition competition since 2011 from 26% to 3.5% and beating a human champion at Go [Jou+18]. At Google, DNNs are applied to a wide range of applications including speech, vision, language, translation, search ranking, and many more. Figure 2.6 illustrates the growth of machine learning at Google.

Neural networks (NN) target brain-like functionality and are based on a simple artificial neuron: a nonlinear function (such as max(0, value)) of a weighted sum of the inputs. These artificial neurons are collected into layers, with the outputs of one layer becoming the inputs of the next one in the sequence. The "deep" part of DNN comes from going beyond a few layers, as the large data sets in the cloud allowed more accurate models to be built by using extra and larger layers to capture higher levels of patterns or concepts [Jou+18].

The two phases of DNNs are *training* (or learning) and *inference* (or prediction), and they refer to DNN model development vs. use [Jou+18]. DNN workloads are further be classified into different categories: convolutional, sequence, embedding-based, multilayer perceptron, and reinforcement learning [MLP18].

Training determines the weights or parameters of a DNN, adjusting them repeatedly until the DNN produces the desired results. Virtually all training is in floating point. During training, multiple learners process subsets of the input training set and reconcile the parameters across the learners either through parameter servers or reduction across learners. The learners typically process the input data set through multiple *epochs*. Training can be done asynchronously [Dea+12], with each learner independently communicating with the parameter servers, or synchronously, where learners operate in lockstep to update the parameters after every step. Recent results show that synchronous training provides better model quality; however, the training performance is limited by the slowest learner [Che+16].

Inference uses the DNN model developed during the training phase to make predictions on data. DNN inference is typically user facing and has strict latency constraints [Jou+17]. Inference can be done using floating point (single precision, half precision) or quantized (8-bit, 16-bit) computation. Careful quantization of models trained in floating point is needed to enable inference without any quality loss compared to the floating point models. Lower precision inference enables lower latency and improved power efficiency for inference [Jou+17].

Three kinds of Neural Networks (NNs) are popular today.

1. *Multi-Layer Perceptrons (MLP)*: Each new layer is a set of nonlinear functions of weighted sum of all outputs (*fully connected*) from a prior one.

2. *Convolutional Neural Networks (CNN)*: Each ensuing layer is a set of nonlinear functions of weighted sums of spatially nearby subsets of outputs from the prior layer, which also reuses the weights.

3. *Recurrent Neural Networks (RNN)*: Each subsequent layer is a collection of nonlinear functions of weighted sums of outputs and the previous state. The most popular RNN is *Long Short-Term Memory* (LSTM). The art of the LSTM is in deciding what to forget and what to pass on as state to the next layer. The weights are reused across time steps.

Table 2.1 describes recent versions of six production applications (two examples of each of the three types of NNs) as well as the ResNet50 benchmark [He+15]. One MLP is a recent version of RankBrain [Cla15]; one LSTM is a subset of GNM Translate [Wu+16b]; one CNN is Inception; and the other CNN is DeepMind AlphaGo [Sil+16].

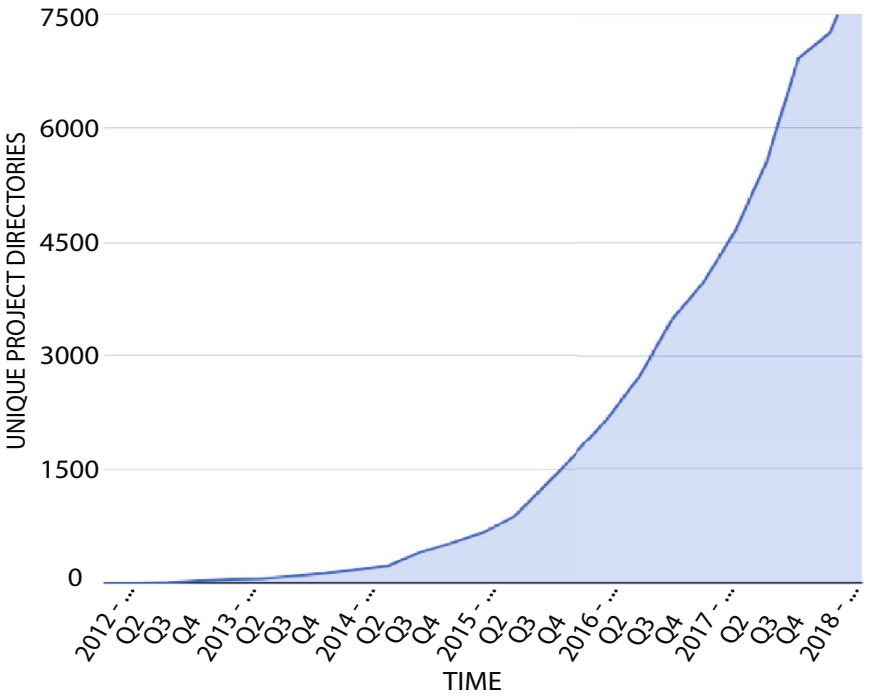

Figure 2.6: Growth of machine learning at Google.

Table 2.1: Six production applications plus ResNet benchmark. The fourth column is the total number of operations (not execution rate) that training takes to converge.

Type of Neural Network	Parameters (MiB)	Training			Inference
		Examples to Convergence	ExaOps to Conv	Ops per Example	Ops per Example
MLP0	225	1 trillion	353	353 Mops	118 Mops
MLP1	40	650 billion	86	133 Mops	44 Mops
LSTM0	498	1.4 billion	42	29 Gops	9.8 Gops
LSTM1	800	656 million	82	126 Gops	42 Gops
CNN0	87	1.64 billion	70	44 Gops	15 Gops
CNN1	104	204 million	7	34 Gops	11 Gops
ResNet	98	114 million	<3	23 Gops	8 Gops

2.5 MONITORING INFRASTRUCTURE

An important part of the cluster-level infrastructure software layer is concerned with various forms of system introspection. Because the size and complexity of both the workloads and the hardware infrastructure make the monitoring framework a fundamental component of any such deployments, we describe it here in more detail.

2.5.1 SERVICE-LEVEL DASHBOARDS

System operators must keep track of how well an internet service is meeting its service level indicators (SLIs). The monitoring information must be very fresh so that an operator (or an automated system) can take corrective actions quickly and avoid significant disruption—within seconds, not minutes. Fortunately, the most critical information needed is restricted to just a few signals that can be collected from the front-end servers, such as latency and throughput statistics for user requests. In its simplest form, such a monitoring system can simply be a script that polls all front-end servers every few seconds for the appropriate signals and displays them to operators in a dashboard. Stackdriver Monitoring [GStaDr] is a publicly available tool which shares the same infrastructure as Google's internal monitoring systems.

Large-scale services often need more sophisticated and scalable monitoring support, as the number of front-ends can be quite large, and more signals are needed to characterize the health of the service. For example, it may be important to collect not only the signals themselves but also their derivatives over time. The system may also need to monitor other business-specific parameters in addition to latency and throughput. The monitoring system supports a simple language that lets operators create derived parameters based on baseline signals being monitored. Finally, the system generates automatic alerts to on-call operators depending on monitored values and thresholds. Fine tuning a system of alerts (or alarms) can be tricky because alarms that trigger too often because of false positives will cause operators to ignore real ones, while alarms that trigger only in extreme cases might get the operator's attention too late to allow smooth resolution of the underlying issues.

2.5.2 PERFORMANCE DEBUGGING TOOLS

Although service-level dashboards help operators quickly identify service-level problems, they typically lack the detailed information required to know *why* a service is slow or otherwise not meeting requirements. Both operators and the service designers need tools to help them understand the complex interactions between many programs, possibly running on hundreds of servers, so they can determine the root cause of performance anomalies and identify bottlenecks. Unlike a service-level dashboard, a performance debugging tool may not need to produce information in real-time for

online operation. Think of it as the data center analog of a CPU profiler that determines which function calls are responsible for most of the time spent in a program.

Distributed system tracing tools have been proposed to address this need. These tools attempt to determine all the work done in a distributed system on behalf of a given initiator (such as a user request) and detail the causal or temporal relationships among the various components involved.

These tools tend to fall into two broad categories: black-box monitoring systems and application/middleware instrumentation systems. WAP5 [Rey+06b] and the Sherlock system [Bah+07] are examples of black-box monitoring tools. Their approach consists of observing networking traffic among system components and inferring causal relationships through statistical inference methods. Because they treat all system components (except the networking interfaces) as black boxes, these approaches have the advantage of working with no knowledge of, or assistance from, applications or software infrastructure components. However, this approach inherently sacrifices information accuracy because all relationships must be statistically inferred. Collecting and analyzing more messaging data can improve accuracy but at the expense of higher monitoring overheads.

Instrumentation-based tracing schemes, such as Pip [Rey+06a], Magpie [Bar+03b], and X-Trace [Fon+07], take advantage of the ability to explicitly modify applications or middleware libraries for passing tracing information across machines and across module boundaries within machines. The annotated modules typically also log tracing information to local disks for subsequent collection by an external performance analysis program. These systems can be very accurate as there is no need for inference, but they require all components of the distributed system to be instrumented to collect comprehensive data. The Dapper [Sig+10] system, developed at Google, is an example of an annotation-based tracing tool that remains effectively transparent to application-level software by instrumenting a few key modules that are commonly linked with all applications, such as messaging, control flow, and threading libraries. Stackdriver Trace [GStaDr] is a publicly available implementation of the Dapper system. XRay, a feature of the LLVM compiler, uses compiler instrumentation to add trace points at every function entry and exit, enabling extremely fine latency detail when active and very low overhead when inactive. For even deeper introspection, the Stackdriver Debugger [GStaDb] tool allows users to dynamically add logging to their programs without recompiling or redeploying.

CPU profilers based on sampling of hardware performance counters have been incredibly successful in helping programmers understand microarchitecture performance phenomena. Google-Wide Profiling (GWP) [Ren+10] selects a random subset of machines to collect short whole machine and per-process profile data, and combined with a repository of symbolic information for all Google binaries produces cluster-wide view of profile data. GWP answers questions such as: which is the most frequently executed procedure at Google, or which programs are the largest users of memory? The publicly available Stackdriver Profiler [GStaPr] product is inspired by GWP.

2.5.3 PLATFORM-LEVEL HEALTH MONITORING

Distributed system tracing tools and service-level dashboards measure both health and performance of applications. These tools can infer that a hardware component might be misbehaving, but that is still an indirect assessment. Moreover, because both cluster-level infrastructure and application-level software are designed to tolerate hardware component failures, monitoring at these levels can miss a substantial number of underlying hardware problems, allowing them to build up until software fault-tolerance can no longer mitigate them. At that point, service disruption could be severe. Tools that continuously and directly monitor the health of the computing platform are needed to understand and analyze hardware and system software failures. In Chapter 7, we discuss some of those tools and their use in Google's infrastructure in more detail.

Site reliability engineering

While we have talked about infrastructure software thus far, most WSC deployments support an important function called "site reliability engineering," different from traditional system administration in that software engineers handle day-to-day operational tasks for systems in the fleet. Such SRE software engineers design monitoring and infrastructure software to adjust to load variability and common faults automatically so that humans are not in the loop and frequent incidents are self-healing. An excellent book by a few of Google's site reliability engineers [Mur+16] summarizes additional principles used by SREs in large WSC deployments.

2.6 WSC SOFTWARE TRADEOFFS

2.6.1 DATA CENTER VS. DESKTOP

Software development in internet services differs from the traditional desktop/server model in many ways.

- *Ample parallelism*: Typical internet services exhibit a large amount of parallelism stemming from both data- and request-level parallelism. Usually, the problem is not to find parallelism but to manage and efficiently harness the explicit parallelism inherent in the application. Data parallelism arises from the large data sets of relatively independent records that need processing, such as collections of billions of web pages or billions of log lines. These very large data sets often require significant computation for each parallel (sub) task, which in turn helps hide or tolerate communication and synchronization overheads. Similarly, request-level parallelism stems from the hundreds or thousands of requests per second that popular internet services receive. These requests rarely involve read-write sharing of data or synchronization across requests. For

example, search requests are essentially independent and deal with a mostly read-only database; therefore, the computation can be easily partitioned both within a request and across different requests. Similarly, whereas web email transactions do modify user data, requests from different users are essentially independent from each other, creating natural units of data partitioning and concurrency. As long as the update rate is low, even systems with highly interconnected data (such as social networking backends) can benefit from high request parallelism.

- *Workload churn*: Users of internet services are isolated from the services' implementation details by relatively well-defined and stable high-level APIs (such as simple URLs), making it much easier to deploy new software quickly. Key pieces of Google's services have release cycles on the order of a couple of weeks compared to months or years for desktop software products. Google's front-end web server binaries, for example, are released on a weekly cycle, with nearly a thousand independent code changes checked in by hundreds of developers—the core of Google's search services is re-implemented nearly from scratch every 2–3 years. This environment creates significant incentives for rapid product innovation, but makes it hard for a system designer to extract useful benchmarks even from established applications. Moreover, because internet services is still a relatively new field, new products and services frequently emerge, and their success with users directly affects the resulting workload mix in the data center. For example, video services such as YouTube have flourished in relatively short periods and present a very different set of requirements from the existing large customers of computing cycles in the data center, potentially affecting the optimal design point of WSCs. A beneficial side effect of this aggressive software deployment environment is that hardware architects are not necessarily burdened with having to provide good performance for immutable pieces of code. Instead, architects can consider the possibility of significant software rewrites to leverage new hardware capabilities or devices.

- *Platform homogeneity*: The data center is generally a more homogeneous environment than the desktop as a target platform for software development. Large internet services operations typically deploy a small number of hardware and system software configurations at any given time. Significant heterogeneity arises primarily from the incentive to deploy more cost-efficient components that become available over time. Homogeneity within a platform generation simplifies cluster-level scheduling and load balancing and reduces the maintenance burden for platforms software, such as kernels and drivers. Similarly, homogeneity can allow more efficient supply chains and more efficient repair processes because automatic and manual repairs benefit from having more experience with fewer types of systems. In contrast, software for desktop systems

can make few assumptions about the hardware or software platform they are deployed on, and their complexity and performance characteristics may suffer from the need to support thousands or even millions of hardware and system software configurations.

- *Fault-free operation*: Because internet service applications run on clusters of thousands of machines—each of them not dramatically more reliable than PC-class hardware—the multiplicative effect of individual failure rates means that some type of fault is expected every few hours or less (more details are provided in Chapter 7). As a result, although it may be reasonable for desktop-class software to assume a fault-free hardware operation for months or years, this is not true for data center-level services: internet services need to work in an environment where faults are part of daily life. Ideally, the cluster-level system software should provide a layer that hides most of that complexity from application-level software, although that goal may be difficult to accomplish for all types of applications.

Although the plentiful thread-level parallelism and a more homogeneous computing platform help reduce software development complexity in internet services compared to desktop systems, the scale, the need to operate under hardware failures, and the speed of workload churn have the opposite effect.

2.6.2 PERFORMANCE AND AVAILABILITY TOOLBOX

Some basic programming concepts tend to occur often in both infrastructure and application levels because of their wide applicability in achieving high performance or high availability in large-scale deployments. The following table (Table 2.2) describes some of the most prevalent concepts. Some articles, by Hamilton [Ham07], Brewer [Bre01], and Vogels [Vog08], provide interesting further reading on how different organizations have reasoned about the general problem of deploying internet services at a very large scale.

Table 2.2: Key concepts in performance and availability trade-offs

Technique	Main Advantages	Description
Replication	Performance and availability	Data replication can improve both throughput and availability. It is particularly powerful when the replicated data is not often modified, since replication makes updates more complex.
Reed-Solomon codes	Availability and space savings	When the primary goal is availability, not throughput, error correcting codes allow recovery from data losses with less space overhead than straight replication.

Sharding (partitioning)	Performance and availability	Sharding splits a data set into smaller fragments (shards) and distributes them across a large number of machines. Operations on the data set are dispatched to some or all of the shards, and the caller coalesces results. The sharding policy can vary depending on space constraints and performance considerations. Using very small shards (or microsharding) is particularly beneficial to load balancing and recovery.
Load-balancing	Performance	In large-scale services, service-level performance often depends on the slowest responder out of hundreds or thousands of servers. Reducing response-time variance is therefore critical. In a sharded service, we can load balance by biasing the sharding policy to equalize the amount of work per server. That policy may need to be informed by the expected mix of requests or by the relative speeds of different servers. Even homogeneous machines can offer variable performance characteristics to a load-balancing client if servers run multiple applications. In a replicated service, the load-balancing agent can dynamically adjust the load by selecting which servers to dispatch a new request to. It may still be difficult to approach perfect load balancing because the amount of work required by different types of requests is not always constant or predictable. Microsharding (see above) makes dynamic load balancing easier since smaller units of work can be changed to mitigate hotspots.

Health checking and watchdog timers	Availability	In a large-scale system, failures often manifest as slow or unresponsive behavior from a given server. In this environment, no operation can rely on a given server to make forward progress. Moreover, it is critical to quickly determine that a server is too slow or unreachable and steer new requests away from it. Remote procedure calls must set well-informed timeout values to abort long-running requests, and infrastructure-level software may need to continually check connection-level responsiveness of communicating servers and take appropriate action when needed.
Integrity checks	Availability	In some cases, besides unresponsiveness, faults manifest as data corruption. Although those may be rare, they do occur, often in ways that underlying hardware or software checks do not catch (for example, there are known issues with the error coverage of some networking CRC checks). Extra software checks can mitigate these problems by changing the underlying encoding or adding more powerful redundant integrity checks.
Application-specific compression	Performance	Often, storage comprises a large portion of the equipment costs in modern data centers. For services with very high throughput requirements, it is critical to fit as much of the working set as possible in DRAM; this makes compression techniques very important because the decompression is orders of magnitude faster than a disk seek. Although generic compression algorithms can do quite well, application-level compression schemes that are aware of the data encoding and distribution of values can achieve significantly superior compression factors or better decompression speeds.

Eventual consistency	Performance and availability	Often, keeping multiple replicas up-to-date using the traditional guarantees offered by a database management system significantly increases complexity, hurts performance, and reduces availability of distributed applications [Vog08]. Fortunately, large classes of applications have more relaxed requirements and can tolerate inconsistent views for limited periods, provided that the system eventually returns to a stable consistent state.
Centralized control	Performance	In theory, a distributed system with a single master limits the resulting system availability to the availability of the master. Centralized control is nevertheless much simpler to implement and generally yields more responsive control actions. At Google, we have tended toward centralized control models for much of our software infrastructure (like MapReduce and GFS). Master availability is addressed by designing master failover protocols.
Canaries	Availability	A very rare but realistic catastrophic failure scenario in online services consists of a single request distributed to a very large number of servers, exposing a program-crashing bug and resulting in system-wide outages. A technique often used at Google to avoid such situations is to first send the request to one (or a few) servers and only submit it to the rest of the system upon successful completion of that (canary) request.
Redundant execution and tail-tolerance	Performance	In very large-scale systems, the completion of a parallel task can be held up by the slower execution of a very small percentage of its subtasks. The larger the system, the more likely this situation can arise. Sometimes a small degree of redundant execution of subtasks can result in large speedup improvements.

2.6.3 BUY VS. BUILD

Traditional IT infrastructure makes heavy use of third-party software components, such as databases and system management software, and concentrates on creating software that is specific to the particular business where it adds direct value to the product offering; for example, as business logic on top of application servers and database engines. Large-scale internet service providers such as Google usually take a different approach, in which both application-specific logic and much of the cluster-level infrastructure software is written in-house. Platform-level software does make use of third-party components, but these tend to be open-source code that can be modified in-house as needed. As a result, more of the entire software stack is under the control of the service developer.

This approach adds significant software development and maintenance work but can provide important benefits in flexibility and cost efficiency. Flexibility is important when critical functionality or performance bugs must be addressed, allowing a quick turnaround time for bug fixes at all levels. It eases complex system problems because it provides several options for addressing them. For example, an unwanted networking behavior might be difficult to address at the application level but relatively simple to solve at the RPC library level, or the other way around.

Historically, when we wrote the first edition of this book, a primary reason favoring build versus buy was that the needed warehouse-scale software infrastructure simply was not available commercially. In addition, it is hard for third-party software providers to adequately test and tune their software unless they themselves maintain large clusters. Last, in-house software may be simpler and faster because it can be designed to address only the needs of a small subset of services, and can therefore be made much more efficient in that domain. For example, BigTable omits some of the core features of a traditional SQL database to gain much higher throughput and scalability for its intended use cases, and GFS falls short of offering a fully Posix compliant file system for similar reasons. Today, such models of scalable software development are more widely prevalent. Most of the major cloud providers have equivalent versions of software developed in-house, and open-source software versions of such software are also widely used (e.g., Kubernetes, Hadoop, OpenStack, Mesos).

2.6.4 TAIL-TOLERANCE

Earlier in this chapter we described a number of techniques commonly used in large-scale software systems to achieve high performance and availability. As systems scale up to support more powerful online web services, we have found that such techniques are insufficient to deliver service-wide responsiveness with acceptable tail latency levels. (*Tail latency* refers to the latency of the slowest requests, that is, the tail of the latency distribution.) Dean and Barroso [DB13] have argued that at large enough scale, simply stamping out all possible sources of performance

variability in individual system components is as impractical as making all components in a large fault-tolerant system fault-free.

Consider a hypothetical system where each server typically responds in 10 ms but with a 99th percentile latency of 1 s. In other words, if a user request is handled on just one such server, 1 user request in 100 will be slow (take 1 s). Figure 2.7 shows how service level latency in this hypothetical scenario is impacted by very modest fractions of latency outliers as cluster sizes increase. If a user request must collect responses from 100 such servers in parallel, then 63% of user requests will take more than 1 s (marked as an "x" in the figure). Even for services with only 1 in 10,000 requests experiencing over 1 s latencies at the single server level, a service with 2,000 such servers will see almost 1 in 5 user requests taking over 1 s (marked as an "o") in the figure.

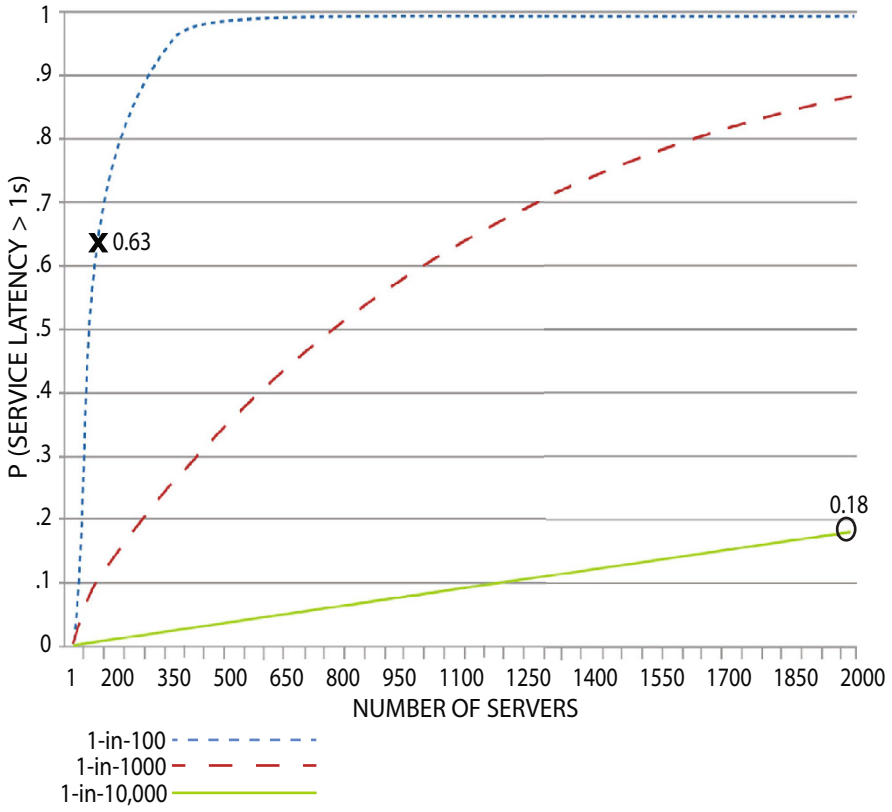

Figure 2.7: Probability of >1 s service level response time as the system scale and frequency of server level high latency outliers varies.

Dean and Barroso show examples of programming techniques that can tolerate these kinds of latency variability and still deliver low tail latency at the service level. The techniques they pro-

pose often take advantage of resource replication that has already been provisioned for fault-tolerance, thereby achieving small additional overheads for existing systems. They predict that tail tolerant techniques will become more invaluable in the next decade as we build ever more formidable online web services.

2.6.5 LATENCY NUMBERS THAT ENGINEERS SHOULD KNOW

This section is inspired by Jeff Dean's summary of key latency numbers that engineers should know [Dea09]. These rough operation latencies help engineers reason about throughput, latency, and capacity within a first-order approximation. We have updated the numbers here to reflect technology and hardware changes in WSC.

Table 2.3: Latency numbers that every WSC engineer should know. (Updated version of table from [Dea09].)

Operation	Time
L1 cache reference	1.5 ns
L2 cache reference	5 ns
Branch misprediction	6 ns
Uncontended mutex lock/unlock	20 ns
L3 cache reference	25 ns
Main memory reference	100 ns
Decompress 1 KB with Snappy [Sna]	500 ns
"Far memory"/Fast NVM reference	1,000 ns (1us)
Compress 1 KB with Snappy [Sna]	2,000 ns (2us)
Read 1 MB sequentially from memory	12,000 ns (12 us)
SSD Random Read	100,000 ns (100 us)
Read 1 MB bytes sequentially from SSD	500,000 ns (500 us)
Read 1 MB sequentially from 10Gbps network	1,000,000 ns (1 ms)
Read 1 MB sequentially from disk	10,000,000 ns (10 ms)
Disk seek	10,000,000 ns (10 ms)
Send packet California→Netherlands→California	150,000,000 ns (150 ms)

2.7 CLOUD COMPUTING

Recently, cloud computing has emerged as an important model for replacing traditional enterprise computing systems with one that is layered on top of WSCs. The proliferation of high speed inter-

net makes it possible for many applications to move from the traditional model of on-premise computers and desktops to the cloud, running remotely in a commercial provider's data center. Cloud computing provides efficiency, flexibility, and cost saving. The cost efficiency of cloud computing is achieved through co-locating multiple virtual machines on the same physical hosts to increase utilization. At a high level, a virtual machine (VM) is similar to other online web services, built on top of cluster-level software to leverage the entire warehouse data center stack. Although a VM-based workload model is the simplest way to migrate on-premise computing to WSCs, it comes with some additional challenges: I/O virtualization overheads, availability model, and resource isolation. We discuss these in more detail below.

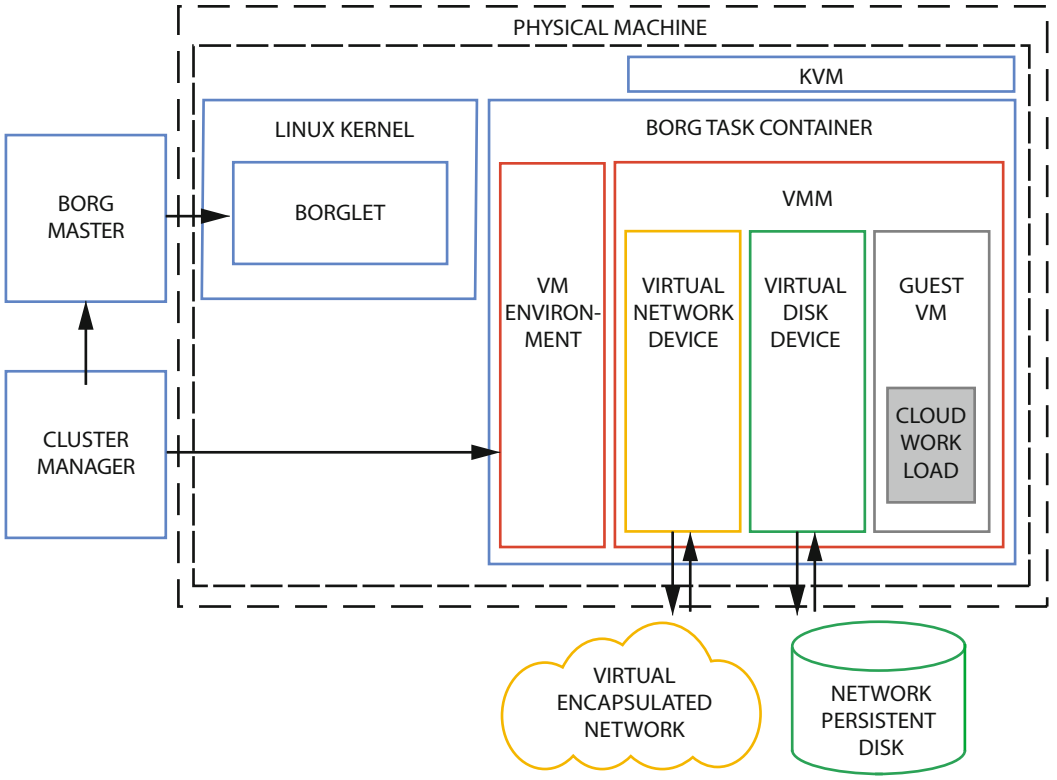

Figure 2.8: Overview of VM-based software stack for Google Cloud Platform workloads.

- *I/O Virtualization*: A VM does not have direct access to hardware resources like local hard drives or networking. All I/O requests go through an abstraction layer, such as *virtio*, in the guest operating system. The hypervisor or virtual machine monitor (VMM) translates the I/O requests into the appropriate operations: For example, storage requests are redirected to the network persistent disk or local SSD drives, and

networking requests are sent through virtualized network for encapsulation. Although I/O virtualization often incurs some performance overhead improvements in both virtualization techniques and hardware support for virtualization have steadily reduced these overheads [Dal+18].

- *Availability model*: Large-scale distributed services achieve high availability by running multiple instances of a program within a data center, and at the same time maintaining N + 1 redundancy at the data center level to minimize the impact of scheduled maintenance events. However, many enterprise applications have fewer users and use older stacks, such as relational databases, where horizontal scaling is often not possible. Cloud providers instead use *live migration* technology to ensure high availability by moving running VMs out of the way of planned maintenance events, including system updates and configurations changes. As a recent example, when responding to the Meltdown and Spectre vulnerabilities [Hen+18]. Google performed host kernel upgrades and security patches to address these issues across its entire fleet without losing a single VM.

  Resource isolation: As discussed earlier, the variability in latency of individual components is amplified at scale at the service level due to interference effects. Resource quality of service (QoS) has proved to be an effective technique for mitigating the impact of such interference. In cloud computing, malicious VMs can exploit multi-tenant features to cause severe contention on shared resources, conducting Denial of Service (DoS) and side-channel attacks. This makes it particularly important to balance the tradeoff between security guarantees and resource sharing.

2.7.1 WSC FOR PUBLIC CLOUD SERVICES VS. INTERNAL WORKLOADS

Google began designing WSCs nearly a decade before it began offering VM-based public clouds as an enterprise product, the Google Cloud Platform [GCP]. During that time Google's computing needs were dominated by search and ads workloads, and therefore it was natural consider the design of both WSCs as well as its corresponding software stack in light of the requirements of those specific workloads. Software and hardware engineers took advantage of that relatively narrow set of requirements to vertically optimize their designs with somewhat tailored, in-house solutions across the whole stack and those optimizations resulted in a company-internal development environment that diverged from that of early public cloud offerings. As the portfolio of internal workloads broadened, Google's internal designs had to adapt to consider more general purpose use cases and as a consequence, the gap between internal workload requirements and those of public clouds has narrowed significantly in most application areas. Although some differences still remain, internal

and external developers experience a much more similar underlying platform today, which makes it possible to bring internal innovations to external users much more quickly than would have been possible a decade ago. One example is the recent public availability of TPUs to Google Cloud Platform customers, discussed more in Chapter 3.

2.7.2 CLOUD NATIVE SOFTWARE

The rising popularity of Cloud Services brought with it a new focus on software that takes full advantage of what cloud providers have to offer. Coupled with the shift toward containers as the vehicle for workloads, the "Cloud Native" ethos emphasizes properties of clouds that are not easily realized in private data centers. This includes things like highly dynamic environments, API-driven self-service operations, and instantaneous, on-demand resources. Clouds are elastic in a way that physical WSCs have difficulty matching. These properties allow developers to build software that emphasizes scalability and automation, and minimizes operational complexity and toil.

Just as containers and orchestrators catalyzed this shift, other technologies are frequently adopted at the same time. Microservices are the decomposition of larger, often monolithic, applications into smaller, limited-purpose applications that cooperate via strongly defined APIs, but can be managed, versioned, tested, and scaled independently. Service meshes allow application operators to decouple management of the application itself from management of the networking that surrounds it. Service discovery systems allow applications and microservices to find each other in volatile environments, in real time.

2.8 INFORMATION SECURITY AT WAREHOUSE SCALE

Cloud users depend on the ability of the WSC provider to be a responsible steward of user or customer data. Hence, most providers invest considerable resources into a sophisticated and layered security posture. The at-scale reality is that security issues will arise, mandating a holistic approach that considers prevention, detection, and remediation of issues. Many commercial software or hardware vendors offer individual point solutions designed to address specific security concerns, but few (if any) offer solutions that address end-to-end security concerns for WSC. The scope is as broad as the scale is large, and serious WSC infrastructure requires serious information security expertise.

At the hardware level, there are concerns about the security of the physical data center premises. WSC providers employ technologies such as biometric identification, metal detection, cameras, vehicle barriers, and laser-based intrusion detection systems, while also constraining the number of personnel even permitted onto the data center floor (for example, see https://cloud.google.com/security/infrastructure/design/). Effort is also invested to ascertain the provenance of hardware and the designs on which it is based. Some WSC providers invest in custom solutions for cryptographic

machine identity and the security of the initial firmware boot processes. Examples include Google's Titan, Microsoft's Cerberus, and Amazon's Nitro.

Services deployed on this infrastructure use encryption for inter-service communication; establish systems for service identity, integrity, and isolation; implement access control mechanisms on top of identity primitives; and ultimately carefully control and audit access to end user or customer data. Even with the correct and intended hardware, issues can arise. The recent Spectre/Meltdown/L1TF issues [Hen+18] are illustrative of the need for WSC providers to maintain defense-in-depth and expertise to assess risk and develop and deploy mitigations. WSC providers employ coordinated management systems that can deploy CPU microcode, system software updates, or other patches in a rapid but controlled manner. Some even support live migration of customer VMs to allow the underlying machines to be upgraded without disrupting customer workload execution.

Individual challenges, such as that of authenticating end user identity, give rise to a rich set of security and privacy challenges on their own. Authenticating users in the face of password reuse and poor password practices at scale, and otherwise preventing login or account recovery abuse often require dedicated teams of experts.

Data at rest must always be encrypted, often with both service- and device-centric systems with independent keys and administrative domains. Storage services face a form of cognitive dissonance, balancing durability requirements against similar requirements to be able to truly, confidently assert that data has been deleted or destroyed. These services also place heavy demands on the network infrastructure in many cases, often requiring additional computational resources to ensure the network traffic is suitably encrypted.

Internal network security concerns must be managed alongside suitable infrastructure to protect internet-scale communication. The same reliability constraints that give rise to workload migration infrastructure also require network security protocols that accommodate the model of workloads disconnected from their underlying machine. One example is Google application layer transport security (ALTS) [Gha+17]. The increasing ubiquity of SSL/TLS, and the need to respond appropriately for ever-present DoS concerns, require dedicated planning and engineering for cryptographic computation and traffic management.

Realizing the full benefit of best-practices infrastructure security also requires operational security effort. Intrusion detection, insider risk, securing employee devices and credentials, and even basic things such as establishing and adhering to best practices for safe software development all require non-trivial investment.

As information security begins to truly enjoy a role as a first-class citizen in WSC infrastructure, some historical industry norms are being reevaluated. For example, it may be challenging for a WSC to entrust critical portions of its infrastructure to an opaque or black-box third-party component. The ability to return machines to a known state—all the way down to the firmware level—in between potentially mutually distrusting workloads is becoming a must-have.