CHAPTER

HYPERSCALABILITY – THE CHANGING FACE OF SOFTWARE ARCHITECTURE

2

Ian Gorton
Northeastern University, United States

2.1 INTRODUCTION

It seems difficult to believe that websites such as Youtube.com (debuted in November 2005) and Facebook.com (public access in 2006) have been around for barely a decade. In 2015 Youtube had more than a billion users, who watched 4 billion videos per day and uploaded 300 hr of video per minute.[1] In 2009, Facebook stored 15 billion photos, occupying 1.5 petabytes (PBs) and at that time growing at a rate of 30 million photos per day. In 2015, Facebook users uploaded 2 billion photos each day, requiring 40 PB of new disk capacity daily.[2]

Traffic and storage magnitudes such as these will not become smaller or easier to deal with in the future. In terms of this rate of growth, Youtube and Facebook are by no means unique. With the imminent explosion of the Internet-of-Things – up to 50 billion new devices are forecast by 2020[3] – the scale of the systems we build to capture, analyze, and exploit this ballooning data will continue to grow exponentially.

The scale of contemporary Internet-based systems, along with their rate of growth, is daunting. Data repositories are growing at phenomenal rates, and new data centers hosting tens of thousands of machines are being built all over the world to store, process, and analyze this data. Societal change driven by these systems has been immense in the last decade, and the rate of innovation is only set to grow. We have truly entered the era of big data.

To address this explosion of data and processing requirements, we need to build systems that can be scaled rapidly with controllable costs and schedules. In this chapter, we refer to such systems as *hyperscalable* systems. Hyperscalable systems can grow their capacity and processing capabilities exponentially to serve a potentially global user base, while scaling linearly the resources and costs needed to deliver and operate the system. An example of a hyperscalable system is Pinterest.com.[4] Pinterest provides a virtual pin board to which users can attach and share content and ideas. From January 2011 to October 2012, their requests load doubled every six weeks to reach tens of billions of page views

[1] http://expandedramblings.com/index.php/youtube-statistics/ viewed October 2015.
[2] http://www.nextplatform.com/2015/05/07/cold-storage-heats-up-at-facebook/.
[3] http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf.
[4] http://highscalability.com/blog/2013/4/15/scaling-pinterest-from-0-to-10s-of-billions-of-page-views-a.html.

Software Architecture for Big Data and the Cloud. DOI: 10.1016/B978-0-12-805467-3.00002-8

each month. During this time they grew from two founders and one engineer to 40 engineers in total – if engineering growth had kept pace with request loads then the engineering team would have been more than 50,000! At the same time, their cloud-based data storage grew by more than $10\times$, and while processing resources grew significantly, they cost only \$52 per hour at peak and as little as \$15 per hour during the night to operate.[5]

Experience building hyperscalable systems has clearly demonstrated that requirements for extreme scale challenge and break many dearly held tenets of software engineering [1]. For example, hyperscale systems cannot be thoroughly system tested before deployment due to their scale and need to run 24×7. Hence new, innovative engineering approaches must be adopted to enable systems to rapidly scale at a pace that keeps up with business and functional requirements, and at acceptable, predictable costs [2].

This chapter is about engineering systems at hyperscale for both cloud and private data center hosted deployments. It describes the characteristics of hyperscale systems, and some of the core principles that are necessary to ensure hyperscalability. These principles are illustrated by state-of-the-art approaches and technologies that are used to engineering hyperscalable systems.

2.2 HYPERSCALABLE SYSTEMS

As a quality attribute description, *scalability* is a widely used term in the software industry. Therefore, before we discuss hyperscalable systems, let's analyze in this section precisely what can be meant by "scalability." We will then build upon this understanding to describe hyper scalable systems.

2.2.1 SCALABILITY

Scalability is widely and intuitively understood in the software engineering and research community as something along the lines of being able to increase a system's capacity as its processing load grows. An example of this from the top of a Google search for "What is scalability?" is from Microsoft's MSDN[6]:

"*Scalability is the ability of a system to expand to meet your business needs. You scale a system by adding extra hardware or by upgrading the existing hardware without changing much of the application.*"

This emphasizes expanding hardware capacity without changing "much" of an application. "Much" seems a rather imprecise term, but the general gist of the meaning of the definition is clear. The software architecture literature, for example, [3], follows along these lines, augmenting the definition by differentiating vertical (bigger machines) and horizontal (more machines) scalability. These are also commonly known as scale up and scale out, respectively.

Other attempts to crystallize a universal definition for scalability exist, focusing on achieving speedups in parallel systems [4], and discussing the different *dimensions* of a problem that are related to a system's requirements that are important when discussing scalability [5–9]. The dimensions discussed in these papers, such as CPUs, memory, network and disk, are interestingly related to com-

[5] http://highscalability.com/blog/2012/5/21/pinterest-architecture-update-18-million-visitors-10x-growth.html.
[6] https://msdn.microsoft.com/en-us/library/aa578023.aspx.

putation, not data. Others cogently describe the influence of processes and people as well as technical design on achieving scalability [26]. These efforts collectively leave little doubt that scalability is a multidimensional problem.

Scale up and scale out are indeed the primary mechanisms for expanding a system's computational capacity. However, these definitions obscure some fundamental complexities concerning both computational scaling, and project context. These essentially revolve around inherent limits in parallelism that exist in applications and limit scalability, and the dimensions of cost and effort associated with achieving scalability requirements. Let's examine these two issues in detail.

2.2.2 SCALABILITY LIMITS

In the fast moving age of computing, 1967 is the equivalent of times when Neanderthals roamed the earth. It is however the year that Amdahl's Law was first presented [10]. Simply stated, Amdahl's law places a theoretical limit on the speedup possible, in terms of latency to complete a task, for a given workload based on the proportional of a task that must be executed sequentially. For example, a data analysis task takes 10 min to execute for a 1 TB data set. The data set can be partitioned and processed in parallel without dependencies between tasks, but the results of each of these parallel tasks need to be merged serially to produce results once the parallel tasks are complete. If the results processing takes 1 min (10% of the runtime), then the overall latency for the task cannot be less than 1 min. As Fig. 2.1 shows, this limits the speedup of the task to $10\times$.

Amdahl's Law was derived in the context of multiprocessor computers and parallel processing for scientific problems. This explains a well-known limitation, namely that it only applies for a fixed size input. This enables the theoretical maximum speedup for a given algorithm to be estimated on a given input data size. In the era of big data, Amdahl's Law is therefore only applicable to problems that have relatively stable input sizes, for example, performing real-time (e.g., 5 s) facial recognition from cameras installed in a large building. After installation, the number of cameras is mostly stable, providing a fixed size problem to solve.

Gustavson's Law [11] reformulates Amdahl's Law to take into account that as computational power increases, the resources can be used to process larger problems in the same time frame. In our facial recognition example, a larger compute cluster would make it possible to process higher resolution or more frequent images from the cameras and still meet the real-time performance constraints. Gustafson's Law is more applicable to the cloud computing environment, where it is possible to provision computational resources on demand to process larger data sets with a latency that meets business needs. This means that as an organization's problem size grows, for example, by acquiring more customers or selling more goods, they can provision more compute power to perform processing and analysis in roughly constant time.

These two laws apply to problems that can be parallelized easily and have approximately $O(n)$ runtimes. For algorithms that are sensitive to input size, for example, that execute with polynomial ($O(n^e)$, $e > 1$) run times, speedups by applying more concurrency to larger problem sizes will be less than that predicted by Gustafson's Law [12]. Many common algorithms (e.g., quicksort) have worst case polynomial run times, as well as many powerful data mining and machine learning techniques. Scaling these requires more craft than simply throwing more compute power at the problem.[7]

[7]http://www.amazon.com/Data-Algorithms-Recipes-Scaling-Hadoop/dp/1491906189.

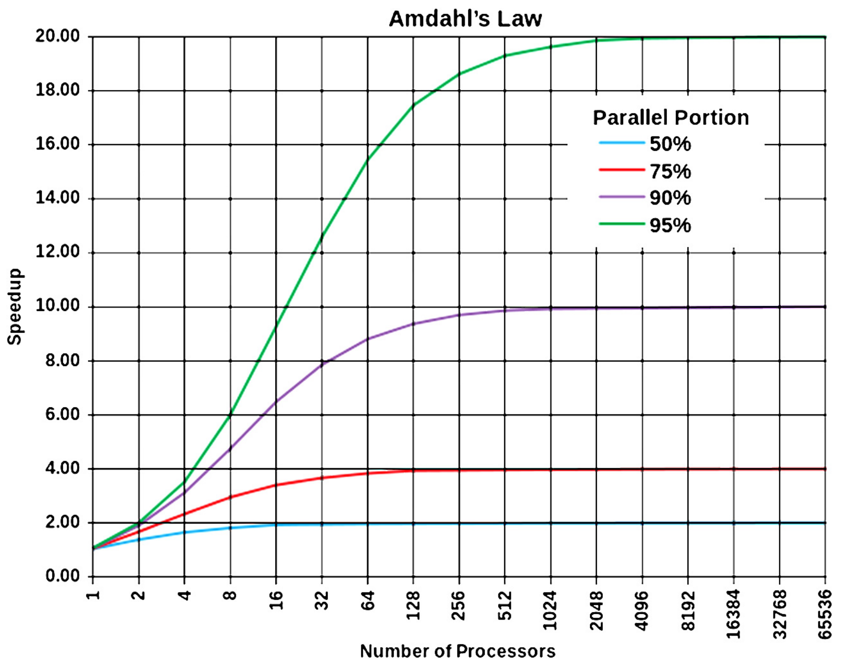

FIGURE 2.1

Amdahl's Law (from https://en.wikipedia.org/wiki/Amdahl%27s_law licensed under CC BY-SA 3.0)

Even if an input data set can be easily partitioned and processed in parallel, the problem of data skew often raises its ugly head. Data skew causes highly variable run times for the tasks that process data partitions in parallel [13]. For example, an application may wish to do a comparison of customer transactions by country. A simple way to process the data in parallel is to allocate the data for all the customers from a single country to a single parallel task, and to provision as many tasks as there are countries. However, if the data set contains 100 million customers in China, and 500 customers in Australia, then the task to process the Chinese customers will take significantly longer than the task to process the Australians. As the customer comparison cannot be completed until the analysis of all countries is complete, the execution time is limited by the time taken to process the data from China.

Data skew is essentially a load imbalance problem, and is commonly observed across different applications, workloads, and business domains. A common approach to dealing with skew is to allocate finer grain partitions to increase parallelism (e.g., process customers in China by province) and reduce task latency. Of course, finer granularity increases overheads and resources, so is not always a panacea. Techniques for both statically and dynamically handling data skew are the subject of ongoing research [14].

In summary, approaches to scaling a problem revolve around scaling up and out computational resources. However, there are fundamental limits to how different types of problems scale, both algorithmically and in terms of their data access patterns. This places an upper bound on how much faster

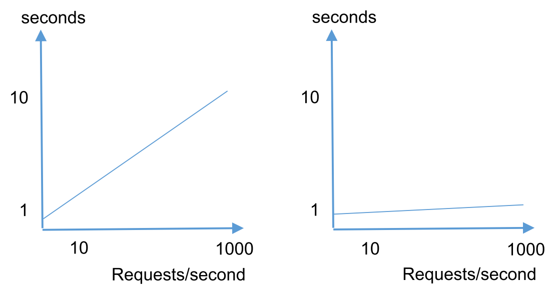

FIGURE 2.2

Scaling an application

a problem can be solved, or what size data sets can be processed in a timely fashion, no matter how many resources can be provisioned to solve a problem.

2.2.3 SCALABILITY COSTS

Unlike engineers from other professions who build and manipulate physical artifacts (e.g., cars, bridges), software engineers primarily manipulate symbolic representations of solutions to problems. These are called programs. While there are physical constraints to consider (e.g., the speed of light for data transmission) when building programs, software is not encumbered by the laws of physics and the complexity of physical construction that other engineering disciplines must deal with.

Perhaps because software is essentially a symbolic artefact with no tangible form for nonengineers to comprehend, we often have expectations of software systems that we would be plainly bizarre for physical artifacts. For example, the Sydney Harbor Bridge was completed in 1932. The traffic volume it carries today far exceeds the capacity it is capable of handling at peak times, causing very significant delays for drivers every single day. However, no one would ever suggest that the capacity of the bridge be doubled or trebled, because the original structure was not built to bear the increased load. Even if it were feasible,[8] the immense costs and work involved, as well as disruption, would be obvious to everyone. With software systems, complexity is hidden to all except those who care to dig into the details.

Let's take a simple hypothetical example to examine this phenomenon in the context of scalability. Assume we have a web-based (web server and database) system that can service a load of 10 concurrent requests with a mean response time of 1 s. We get a business requirement to scale up this system to handle 1000 concurrent requests with the same response time. Without making any changes, a simple test of this system reveals the performance shown in Fig. 2.2 (left). As the request load increases, we see the mean response time steadily grow to 10 s with the projected load. Clearly, this is not scalable and cannot satisfy our requirements in its current deployment configuration.

[8]The story of the expansion of the Auckland Harbor Bridge is instructive – https://en.wikipedia.org/wiki/Auckland_Harbour_Bridge#.27Nippon_clip-ons.27.

Clearly, some engineering effort is needed in order to achieve the required performance. Fig. 2.2 (right) shows the system's performance after it has been modified. It now provides the specified response time with 1000 concurrent requests. Hence we have successfully scaled the system.

A major question looms though, namely, how much effort was required to achieve this performance? Perhaps it was simply a case of running the web server on a more powerful (virtual) machine. Administratively performing such a reprovisioning on a cloud might take 30 min at most. Slightly more complex would be reconfiguring the system to run multiple instances of the web server to increase capacity. Again, this should be a simple, low cost configuration change for the application. These would be excellent outcomes.

However, scaling a system isn't always so easy. The reasons for this are many and varied, but here are some possibilities:

1. The database becomes less responsive with 1000 requests, requiring an upgrade to a new machine.
2. The Web server generates a lot of content dynamically and this reduces response time under load. A possible solution is to alter the code to cache content and reuse it from the cache.
3. The request load creates hot spots in the database when many requests try to access and update the same records simultaneously. This requires a schema redesign and subsequent reloading of the database, as well as code changes to the data access layer.
4. The web server framework that was selected emphasized ease of development over scalability. The model it enforces means that the code simply cannot be scaled to meet the request load requirements, and a complete rewrite is required.

There's a myriad of other potential causes, but hopefully these illustrate the increasing effort that might be required as we move from possibility (1) to possibility (4).

Now let's assume option (1), upgrading the database server, requires 15 hr of effort and a few thousand dollars for a new server. This is not prohibitively expensive. And let's assume option (4), a rewrite of the web application layer, requires 10,000 hr of development due to implementing in a new language (e.g., Java instead of Ruby). Options (2) and (3) fall somewhere in between options (1) and (4). The cost of 10,000 hr of development is seriously significant. Even worse, while the development is underway, the application may be losing market share and hence money due to its inability to satisfy client requests loads. These kinds of situations can cause systems and businesses to fail.

This simple scenario illustrates how the dimensions of cost and effort are inextricably tied to scalability as a quality attribute. If a system is not designed intrinsically to scale, then the downstream costs and resources of increasing its capacity in order to meet requirements may be massive. For some applications, such as Healthcare.gov,[9] these costs are borne and the system is modified eventually to meet business needs. For others, such as Oregon's health care exchange,[10] an inability to scale rapidly at low cost can be an expensive ($303 million) death knell.

We would never expect that someone would attempt to scale up the capacity of a suburban home to become a 50 floor office building. The home doesn't have the architecture, materials and foundations for this to be even a remote possibility without being completely demolished and rebuilt. Similarly, we shouldn't expect software systems that do not employ scalable architectures, mechanisms, and

[9]http://www.cio.com/article/2380827/developer/developer-6-software-development-lessons-from-healthcare-gov-s-failed-launch.html.

[10]http://www.informationweek.com/healthcare/policy-and-regulation/oregon-dumps-failed-health-insurance-exchange/d/d-id/1234875.

technologies to be quickly changed to meet greater capacity needs. The foundations of scale need to be built in from the beginning, with the recognition that the components will evolve over time. By employing design and development principles that promote scalability, we are able to more rapidly and cheaply scale up systems to meet rapidly growing demands. Systems with these properties are hyper scalable systems.

2.2.4 HYPERSCALABILITY

Let's explore the quality attribute of hyperscalability through another simple example. Assume you become the lead engineer for a system that manages a large data collection that is used for both high performance and historical analysis. This might be an Internet-scale application that collects and mines data from user traffic for marketing and advertising purposes. Or it might be a data collection instrument such as a high resolution radio telescope that is constantly scanning the universe and captures data for astronomical analysis and exploration. The exact application doesn't matter for our purposes here.

In your first meeting with the existing technical team and system stakeholders, the current system status is described and future growth projections discussed. The data collection is currently 1 PB in size, and is projected to double in size every 4 months (because of increased user traffic, higher resolution instrumentation, etc.). This means in three years (9×4 months), the data collection will be 2^9, or 512 PB in size. At the end of the meeting, you are tasked with projecting out anticipated operational, development and maintenance resources needed to facilitate this exponential data growth rate. The current budget is $2 million. What budget is needed in 3 yr?

Fig. 2.3 plots various budget growth rates over this two year period. The y-axis represents millions of dollars on a logarithmic scale, with each 4 months period represented on the x-axis. If the budget grows at the same exponential rate as the data, you will need $1024 million in 3 yr. Good luck selling that to your stakeholders! If your budget grows by a constant $500K for each doubling in data size, you will only need a budget of $6.5 million in 3 yr. The graph also shows budget growth rates of 25% and 50% every 4 months, and a growth rate of constant $1 million per 4 months. Clearly, the system sponsors are likely to be more responsive to projections for constant, linear growth. This decouples the costs from the absolute data size that your system manages.

This leads us to a definition of hyperscalable systems:

Hyperscalable systems exhibit exponential growth rates in computational resources while exhibiting linear growth rates in the operational costs of resources required to build, operate, support, and evolve the required software and hardware resources.

In this context:

- **Computational resources** refer to everything required to execute the system. This includes data storage, both on- and offline, processing resources such as CPU nodes and network bandwidth (from dedicated data centers or shared cloud based), and physical resources such as data center construction and power consumption. In big data systems, growth in data size is typically the driving dimension of exponential growth.
- **Operational costs** are all those that are associated with utilizing the computational resources. These include software development costs for evolving existing and building new features, data management platform costs (software and hardware), operations, training and customer support costs, and cost of acquisition of processing nodes (purchase, cloud-based).

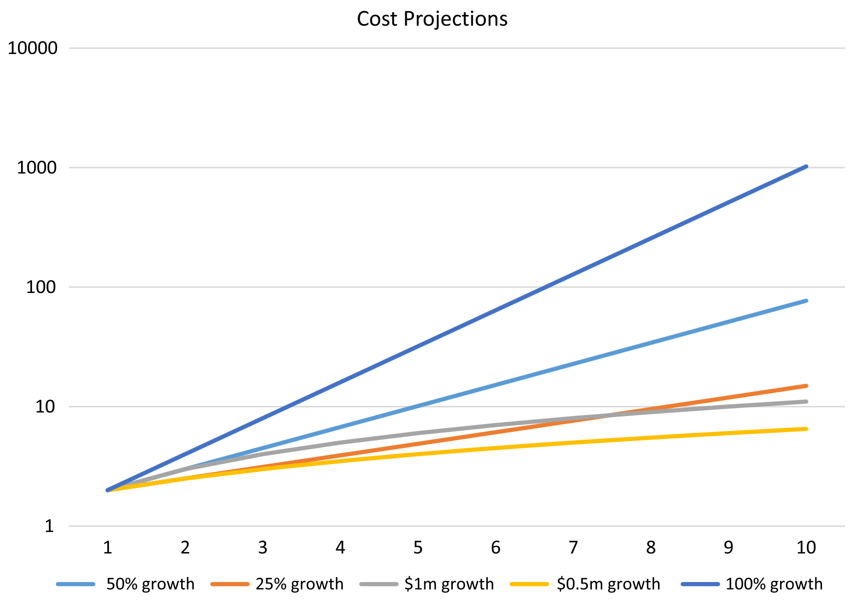

FIGURE 2.3

Cost growth projections in $ millions (y-axis) over a 3 yr period (x-axis in 4 month increments)

The need to scale rapidly therefore is foundational to hyperscalable systems. If they are unable to scale to meet business needs, or achieve scalability but at costs that will become unsustainable, they will fail. Scale is the primary influence on design and engineering, as it is the key quality needed for survival. When design tradeoffs are inevitably needed, the option that promotes scaling wins. Other qualities must of course be achieved too – for example, availability and security – but scale drives decisions forward.

The remainder of this chapter discusses some software engineering principles and approaches that are fundamental to building hyperscalable systems.

2.3 PRINCIPLES OF HYPERSCALABLE SYSTEMS

Hyperscalable systems are immense endeavors, and likely represent some of the most complex achievements that humans have ever built. As with any systems of such complexity, a number of principles underpin their engineering. In this section, we describe some general principles that hold for any hyperscalable system. These principles can help designers continually validate major design decisions across development iterations, and hence provide a guide through the complex collection of design trade-offs that all hyperscale systems require.

2.3.1 AUTOMATE AND OPTIMIZE TO CONTROL COSTS

There are typically two major costs in engineering software systems, namely people and resources (e.g., CPU, disk, network – see above definition of hyperscale systems). As a system's scale increases, inevitably more engineers are needed to design and deploy software, and more compute, network, storage, power, and facility costs are needed to support the system. These consequently are the major source of costs that must be controlled as systems grow.

There are two basic approaches to controlling these costs. First, automation of a whole spectrum engineering and deployment tasks is fundamental, as this reduces human costs and enables organizations to keep team sizes small. Second, optimization of software and data storage makes it possible to reduce the deployment footprint of system while maintaining its capacity to handle requests. As the scale of a system grows, optimizations result in significant absolute costs savings, as we will describe in the following sections.

2.3.1.1 Automation

Modern software engineering is replete with automated processes, for example, automated build and testing. For hyperscale systems, these approaches are equally important. Other characteristics of engineering at scale, however, require further automation. Let's examine a variety of these below.

When you upgrade the features of a site such as netflix.com, it's not possible to bring the whole system down, deploy the new version and start it up again. Internet systems aim for 100% availability. They must be upgraded and tested while live. Automated unit and subsystems test pipelines are still crucial to validate functional correctness, but it is pragmatically impossible to test every component under the load that they will experience in production (imagine the costs and time of setting up a test environment with 100 PB of data and simulating the load of 10 million concurrent users). The only way to test is to deploy the new features in the live system and monitor how they behave.

Testing in a live system is something that goes against established software engineering practices, but this is routinely done in hyperscale systems without bringing down the application. This is achieved through deploying a new component side by side with the replaced component, and directing only a portion (e.g., 1%) of the request load to the new component. By monitoring the behavior of the new functions, errors can be quickly and automatically detected, and the component can be undeployed and fixed if needed. If the component behaves correctly under a light load, the load can be gradually increased so that the performance of the new code under more realistic loads can be tested. If all goes smoothly, the new component will process the full request load and the old version is completely replaced [15].

Automated testing can be extended beyond single component testing to proactive failure injection for the whole system. The primary example of such automation is Netflix's Simian Army.[11] This contains a collection of *monkeys* that can be configured to cause failures in a running system. This makes it possible to discover unanticipated system-wide failure modes in a controlled manner, before these failures are inevitably triggered by the system's production load.

Examples of the Simian Army are:

[11] http://techblog.netflix.com/2011/07/netflix-simian-army.html.

- Chaos Monkey: The Chaos Monkey can be configured to randomly kill live virtual machines in a running system. This simulates random failures and enables an engineering team to validate the robustness of their systems.
- Latency Monkey: This introduces delays into the communications between the components in a system, making it possible to test the behavior of the calling components when they experience delays in their dependent components. This is particularly useful when testing the fault-tolerance of a new component by simulating the failure of its dependencies (by introducing long latencies), without making these dependencies unavailable to the rest of the system.
- Security Monkey: This can be configured to test for security vulnerabilities, and check the validity of certificates so that they do not expire unexpectedly.

Testing is not the only engineering task that benefits immensely from automation. Here are some more examples.

Applications that acquire and persist data at rapid rates must continually grow their storage resources while maintaining consistent access times to data. This requires that new database servers be continually added at a rate sufficient to handle data growth and that the data be sharded and balanced across the servers so that contention and hotspots are minimized. Manually rebalancing data across many shards is obviously time consuming and infeasible at scale. Consequently, many scalable database technologies will automatically rebalance data across servers based on a trigger such as adding a new server or detecting imbalance across existing server nodes. Consistent hashing schemes [16] provide the most efficient mechanism to perform this rebalancing as they minimize the amount of data that must be moved between shards as the database key space is repartitioned across available server nodes. An automated, efficient database expansion and rebalancing mechanism is fundamental requirement for hyperscale systems.

Many systems experience cyclic request loads, where peak loads are often an order of magnitude (or more) greater than average loads. Traditionally this has required a system to over-provision processing resources to be able to handle peaks loads. During average or low load periods, this means many of the resources remain idle. For such cyclic loads, the canonical example being Netflix's predictable diurnal load cycle,[12] it is possible to create an elastic solution that handles peaks by temporarily provisioning new cloud-based resources as required, and tearing these down to save costs once the peaks are over. Elastic solutions can be either customized to an application to detect growing and decreases load phases and act accordingly, or can exploit cloud-provider approaches such as AWS' Elastic Load Balancing.[13] By only paying for processing resources when needed, resource costs can be driven down to approximate the cost of the resources needed to process the average load.

2.3.1.2 Optimization

Premature optimization has been seen as an anathema in software engineering for 40 yr or more [17]. However, at hyperscale, even relatively small optimizations can result in significant cost savings. As an example, if by optimizing the web server layer, an application can reduce the number of web servers needed from 1000, to 900, this will both save 10% operational costs and create capacity to handle

[12]http://techblog.netflix.com/2012/01/auto-scaling-in-amazon-cloud.html.
[13]https://aws.amazon.com/elasticloadbalancing/.

increased request loads with 1000 servers. Optimization is therefore fundamental to developing hyper scale systems, as it enables a system to "do more with less."

A well-known example of effective optimization is Facebook's work on a translating PHP to C++, known as HPHPc [18]. Facebook extensively used PHP to build their website. Using HPHPc, the PHP is converted to C++, compiled and executed by the VM (rather than interpreting PHP opcodes). This approach provided up to 6× increased throughput for web page generation. HPHPc was recently replaced by a virtual machine-based solution known as HHVM,[14] and this now provides better performance than the original HPHPc. These innovations enable Facebook to handle greater traffic volumes with less resources, and hence drive down operational costs. Of course, for many organizations, undertaking an effort to write a compiler or virtual machine to optimize their applications is not feasible. Importantly though, the major Internet companies commonly open source the technologies they develop, making it possible for the community to benefit from their innovations. Examples are from Netflix[15] and LinkedIn.[16]

Algorithmic optimization is also fundamental at hyperscale. Linear or near linear complexity algorithms will provide faster and more consistent performance and utilize less resources. This becomes important as the size of the problem increases. Many problems can be amenable to approximate algorithms [19], which sample typically a random subset of the data set and produce results that are within a known bound (the confidence level) of the optimal result. For example, processing a random sample of tweets to understand trending topics, or a random sample of sales from supermarkets spread across the country are algorithms that are amenable to approximation. Approximate algorithms are also known as sublinear algorithms.

2.3.2 SIMPLE SOLUTIONS PROMOTE SCALABILITY

Most of us are taught at an early age that if a deal sounds too good to be true, it probably is. Common sense tells us investments that are guaranteed to grow at 100 percent a year are almost certainly bogus or illegal, so we ignore them. Unfortunately, when building scalable software systems, we commonly see common sense tossed out of the window when competing design alternatives and technologies are evaluated as candidates for major components of big data systems.

Let's take a simple example: Strong consistency in databases is the bedrock of transactional systems and relational databases. Implementing strong consistency, however, is expensive. This is especially true in distributed databases due to the necessary overheads such as locking, communication latencies, complex failure and recovery modes associated with distributed commit protocols. To build highly scalable and available systems, the NoSQL [1] database movement has consequently weakened the consistency models we can expect from databases. This trend has occurred for a good reason: weak consistency models are inherently more efficient to implement because the underlying mechanisms required are simpler. Weak consistency models tolerate inconsistency in the face of unavailable replicas, trading off response time and hence scalability against consistency. The inevitable trade-off is that the burden of handling replica inconsistencies is placed on the programmer, with varying degrees of

[14] http://hhvm.com/.
[15] https://netflix.github.io/.
[16] https://engineering.linkedin.com/open-source.

support depending on the precise mechanisms used in the database (e.g., vector clocks, versioning, timestamps).

In response, relational databases and the collection of NewSQL[17] technologies are now turning to new implementation models that provide strong consistency. These solutions "*seek to provide the same scalable performance of NoSQL systems for online transaction processing* (*read-write*) *workloads while still maintaining the ACID guarantees of a traditional database system*."

This approach sounds attractive, and some of the open source technologies that exploit main memory and single-threading such as VoltDB[18] show immense promise. But fundamentally, achieving strong consistency requires more complexity, and as the scale of the problem grows, it is almost certainly not going to scale as well as weak consistency models. Even advanced technologies such as Google's F1[19] that utilized GPS/atomic clocks, suffer from relatively high write latencies in order to maintain consistency.

As always however, there's no free lunch. Simpler, weak consistency models will give your application greater scalability, but there are trade-offs. You probably have to de-normalize your data model and, hence, manage any duplication this introduces. Application code has to handle the inevitable conflicts that arise with weak consistency models when concurrent writes occur at different replicas for the same database key. But, if your data and workload are amenable to a weak consistency model (and many are, even ones we think of as needing strong consistency[20]), it will be a path to scalability. This principle is depicted below in Fig. 2.4.

Another example of the second principle is the scalability of messaging or publish subscribe frameworks. Most of these offer the option of reliable messaging by persisting message queues so that they can survive node failure. Persistence requires writing messages to disk, and hence is inevitably going to be slower and less scalable than a memory-based queue. Memory-based queues are susceptible to message loss, but will provide significantly better throughput and capacity than their persistent counterparts.

There is one more key point to remember. Even though one design mechanism may be fundamentally more scalable than another, the implementation of the mechanism and how you utilize it in applications, determines the precise scalability you can expect in your system. Poorly implemented scalable mechanisms will not scale well, and from experience these are not uncommon. The same applies to inappropriate usage of a scalable mechanism in an application design, such as trying to use a batch solution like Hadoop for real-time querying.

Adhering to this principle requires thinking about the fundamental distributed systems and database design principals and mechanisms that underpin designs. Even simple rules of thumb[21] can be enormously beneficial when considering how a design may scale. Ignoring this principal can lead to systems that are beautifully engineered to satisfy functional and quality attribute requirements, but are unable to scale to meet ever growing demands.[22]

[17] http://en.wikipedia.org/wiki/NewSQL.
[18] https://voltdb.com/.
[19] http://research.google.com/pubs/pub38125.html.
[20] https://cloud.google.com/datastore/docs/articles/balancing-strong-and-eventual-consistency-with-google-cloud-datastore/.
[21] http://perspectives.mvdirona.com/2009/10/17/JeffDeanDesignLessonsAndAdviceFromBuildingLargeScaleDistributed Systems.aspx.
[22] http://www.theregister.co.uk/2014/02/19/some_firstwave_big_data_projects_written_down_says_deloitte/.

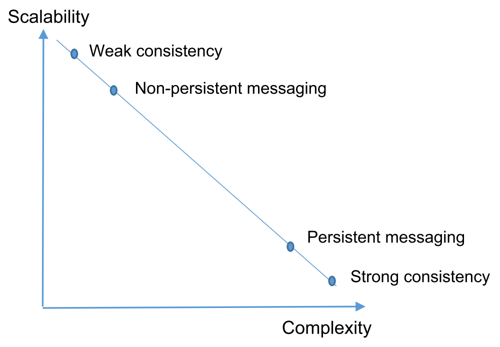

FIGURE 2.4

Scalability versus complexity

2.3.3 UTILIZE STATELESS SERVICES

State management is a much debated and oft misunderstood issue. Many frameworks, for example, the Java Enterprise Edition (JEE), support managing state in the application logic tier by providing explicit abstractions and application programming interfaces (APIs) that load the required state from the database into service instance variables, typically for user session state management. Once in memory, all subsequent requests for that session can hit the same service instance, and efficiently access and manipulate the data that's needed. From a programming perspective, stateful services are convenient and easy.

Unfortunately, from a scalability perspective, stateful solutions are a poor idea for many reasons. First, they consume server resources for the duration of a session, which may span many minutes. Session lengths are often unpredictable, so having many (long-lived) instances on some servers and few on others may create a load imbalance that the system must somehow manage. If a system's load is not balanced, then it has underutilized resources and its capacity cannot be fully utilized. This inhibits scalability and wastes money.

In addition, when sessions do not terminate cleanly (e.g., a user does not log out), an instance remains in memory and consumes resources unnecessarily before some inactive timeout occurs and the resources are reclaimed. Finally, if a server becomes inaccessible due to failure or a network partition, you need some logic, somewhere, to handle the exception and recreate the state on another server.

As we build systems that must manage many millions of concurrent sessions, stateful services become hard to scale. Stateless services, where any service instance can serve any request in a timely fashion, are the scalable solution. There main approach to building stateless systems requires the client to pass a secure session identifier with each request. This identifier becomes the unique key that identifies the session state that is maintained by the server in a cache or database. This state is accessed as needed when new client requests for the session arrive. This allows client request to be served by any stateless server in a replicated server farm. If a server fails while servicing a request, the client can reissue the request to be processed by another server, which leverages the shared session state. Also, new server nodes can be started at any time to add capacity. Finally, if a client becomes inactive on a session, the state associated with that session can simple be discarded (typically based on some timeout value), and the session key invalidated.

As examples, RESTful interfaces are stateless and communicate conversational state using hypermedia links [20]. Netflix's hyperscalable architecture is built upon a foundation of stateless services,[23] and Amazon's AWS cloud platform promotes stateless services to deliver scalability.[24]

2.3.4 OBSERVABILITY IS FUNDAMENTAL TO SUCCESS AT HYPERSCALE

The term *observability* defines the capabilities that make it possible to monitor, analyze, and both proactively and reactively respond to events that occur at runtime in software system. As systems evolve towards hyperscale, it's essential to observe and reason about changes in behavior so that the system can be operated and evolved reliably. The adage of "you can't manage what you don't monitor" is especially true for complex, distributed systems that have an overwhelming number of moving parts, both hardware and software, that interact with each other in many subtle and unanticipated ways.

Here's a simplified example of the problems that can arise in large scale systems. Two separately developed, completely independent business transactions were providing the expected response times when querying a horizontally partitioned database. Suddenly, one transaction slowed down, intermittently, making it occasionally nonresponsive to user needs. Extensive investigations over several days, including detailed logging in production and attempts to recreate the situation in test, eventually led to identifying the root cause. Essentially, periodic and brief request spikes for one of the transactions were overloading a small number of database nodes. During these overload conditions, when the other transaction was invoked, it attempted to read data using a secondary index that was distributed across all nodes. These secondary index reads from the overloaded nodes were taking tens of seconds to respond, leading to unacceptable latencies for those transactions.

How could observability have helped discover the root cause of this problem more quickly? If the developers could have analyzed performance data to visualize transaction volumes for the first transaction against latencies for the second, it would have been immediately obvious that there was a correlation. This would have highlighted the areas of the code that should be investigated, as it was this subtle, unanticipated interaction that was the root cause of the high transaction latencies. Observability in the applications would have recorded transaction latencies and made this data queryable for both real-time and post-mortem analysis.

Detailed performance data at the business transaction level doesn't come for free from databases or web and applications servers. Capturing the necessary performance data to perform this type of analysis requires:

1. Applications to be instrumented with application-relevant measures for observability,
2. A data collection capability to capture and store observability data from the distributed components of the application,
3. Powerful analysis capabilities for developers and operators to gain rapid insights from observability data.

By necessity, Internet companies operating at hyperscale have built their own observability solutions. These solutions are extensive and powerful, and have been built at considerable cost, specifically

[23] https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.
[24] http://highscalability.com/blog/2010/8/16/scaling-an-aws-infrastructure-tools-and-patterns.html.

for each's operational environments. Twitter's solution provides an excellent general blueprint for observability,[25] and Netflix gives a comprehensive discussion of the requirements for a hyperscale observability framework.[26] While these efforts help other organizations with the design of an observability capability, they place the burden of detailed design and implementation on each organization building a hyperscalable system. In a massively distributed system, this burden can be enormous both in cost, effort, and risk.

In recent work, we have taken the conceptual architecture described by Twitter and built a reference implementation for a model-driven observability framework. Model-driven approaches facilitate rapid customization of a framework and eliminate custom code for each deployment, hence reducing costs and effort. In our initial experiments, this framework has been able to efficiently collect and aggregate runtime performance metrics in a big data system with 1000s of storage nodes [21]. The project includes:

1. A model-driven architecture, toolset, and runtime framework that allows a designer to describe a heterogeneous big data storage system as a model, and deploy the model automatically to configure an observability framework.
2. A reference implementation of the architecture, using the open source Eclipse package to implement the model-driven design client, the open source *collectd* package to implement the metric collection component, and the open source *Grafana* package to implement the metrics aggregation and visualization component.
3. Performance and availability results from initial experiments, using the reference implementation.

The initial metamodel and implementation focuses on *polyglot persistence* [1], which employs multiple heterogeneous data stores (often NoSQL/NewSQL) within a single big data system. The reference implementation[27] is suitable for further research to address the challenges of observability in big data systems.

The architecture uses model-driven engineering [22] to automate metric collection, aggregation, and visualization. The main run time elements of the observability system architecture are shown in the top-level component and connector diagram in Fig. 2.5. There are two clients, one for each of the main user roles, *modeling* and *observing*, discussed above. The *Server Tier* includes the *Metric Engine*, which implements dynamic metric aggregation and handles concerns related to dependability of connections to *Collection Daemons*. The *Server Tier* also includes the *Grafana Server*, which handles metric visualization. The *Model Handler* in the *Server Tier* propagates changes to the design-time model, and the *Notification Server* augments the interactive metric visualization with automated notification of user-defined exception conditions.

The *Storage Tier* provides persistent storage of metric streams and notifications. All metrics for each database are stored as a time series to facilitate visualization and analysis. Metrics are stored with metadata to enable dynamic discovery of the metrics. This is necessary to accommodate changes in monitoring configurations after an existing model has been upgraded and deployed as a new version.

The *Metric Monitoring Tier* uses *Observability Daemons* on each database node to collect metrics from the local database instance and operating system. The daemons exploit database-specific and operating system APIs to periodically sample metrics and forward these to the *Metric Engine*.

[25] https://blog.twitter.com/2013/observability-at-twitter.

[26] http://techblog.netflix.com/2014/01/improving-netflixs-operational.html.

[27] Available at https://github.com/johnrklein/obs-prototype.

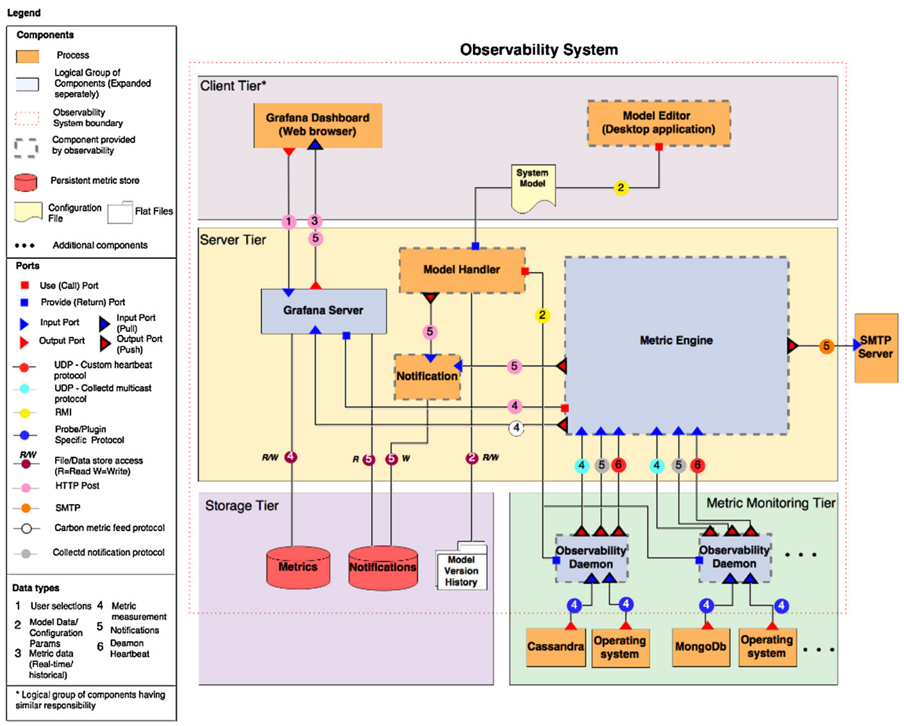

FIGURE 2.5

Observability system architecture (component and connector view)

We performed a series of stress tests on a deployed instance of the observability framework executing on AWS and managing 10,000 simulated database nodes. The metrics collection interval was configured initially at 30 s, and every 5 min was reduced by 5 s. The system operated normally until the sample frequency reached 15 s. At this point, some metrics were not written to disk. This situation continued to deteriorate as we reduced the sampling interval to 5 s. The framework continued to operate but data loss grew.

Examining execution traces from these tests, we saw the CPU, memory, and network utilization levels remained low. Disk writes, however, grew to a peak of 32.7 MB/s. This leads us to believe that the Whisper database[28] in the Grafana server was unable to sustain this relatively heavy write load. This clearly shows that any observability solution for hyperscale systems must be able to sustain heavy write loads. Utilizing a distributed, write-oriented database such as Cassandra or a high throughput file system should provide the write performance required, and we intend to investigate this in future work.

[28] http://graphite.wikidot.com/whisper.

2.4 RELATED WORK

The software engineering research community mainly focuses on scalable systems in two subareas of the discipline, namely Systems of Systems (SoSs) and Ultra large Systems (ULSs). We'll discuss these briefly below.

The term Systems of Systems was coined by Maier [23] in 1996. The main focus was to examine and define the characteristics of systems that grow in scale and complexity through the integration of several independently developed subsystems. Maier defined a set of characteristics that an SoS should exhibit, namely:

- Subsystems are independently useful and can be acquired and operated stand alone, independent of the SoS they can participate in.
- The SoS evolves over time, gradually incorporating new elements and new features in each element.
- Major functionalities of the SoS are achieved through interactions across many subsystems.
- Subsystems are geographically distributed and interact through exchanging messages and information.

The terminology of SoS emerged from Maier's work in the systems engineering domain, with a specific focus on defense systems (a telltale sign of this is the use of the word *acquired* in the above – defense agencies typically acquire (i.e., buy) systems rather than build them). System engineers build incredible complex artifacts, almost always involving physical elements such as a satellite systems or aircraft or submarines. There is considerable software complexity to consider, but this is not the focus of system engineers, or the original target of the SoS work. While the term has been adopted by a small pocket of the software engineering community involved in large scale (typically defense) systems, it is not a term that had gained any traction in the software industry or research community at large. In many ways, the terms Enterprise Application Integration (EAI) and Service Oriented Architecture (SOA) supplanted the need for SoS to be fruitfully considered in software engineering.

The Ultra Large Scale systems project [24] (ULSs) from the Software Engineering Institute built on Maier's work, but sharpened it to have a more specific software and scale focus. While their original target domain was again defense systems, the work clearly articulated the existence of ULSs in domains such as smart cities, the electrical power grid, and intelligent transport systems. The key features of ULSs can be summarized as:

- Decentralization in multiple dimensions – development, operations, data, evolution and design;
- Complex, ever evolving and inevitably conflicting requirements;
- Continuous evolution of the heterogeneous operational system. It can't be stopped to upgrade;
- System usage drives new insights and evolution;
- Software and hardware failures are the norm.

The original ULS book was published in 2006, just as the revolution in Internet scale systems was starting to gather pace. Undoubtedly, the ULS work was visionary, and the above characteristics apply to the systems built today by Google, Amazon, Facebook, Netflix and the like. In many ways, ULSs have rapidly become commonplace in today's world, and their complexity will only continue to grow.

The ULS authors defined a research agenda and roadmap, and the following quote nicely encapsulates their intent:

"*We need a new science to support the design of all levels of the systems that will eventually produce the ULS systems we can envision, but not implement effectively, today*."

Ten years later, we all use ULSs every day through our web browser. They can be built, operated and continuously evolved. The Internet scale revolution became the catalyst for the creation of new technologies, tools, and development methodologies alluded to by the "new science" in the above quotation. In addition, business imperatives, not a focus of the ULS work, forced the economic considerations of scale to rapidly come to the forefront and shape the inventions that have become foundational to engineering the hyperscale systems that are the subject of this chapter.

For observability, we know of no solutions that can be adopted and rapidly tailored for deployment at hyperscale. General distributed system monitoring tools have existed for many years. Commercial tools such as AppDynamics[29] are comprehensive products, but like there commercial counterparts in the database arena, license costs rapidly become an issue as systems scale across hundreds and thousands of servers. Open source equivalents such as Nagios[30] and Ganglia[31] are also widely used, and a useful comparison of technologies in this space can be found in [25]. Adopting these technologies and tailoring them to highly heterogeneous execution environments to observe application-relevant measures, as well as making them operate at the scale required by the next generation of big data applications, will however represent a major challenge for any development organization.

2.5 CONCLUSIONS

The ability to rapidly scale at low costs is a defining characteristic of many modern applications. Driven by ever growing data volumes and needs for valuable discoveries from analyzing vast data repositories, the challenges of building these systems will only increase. Systems that are unable to economically scale are destined for limited impact and short life times.

Hyperscale systems are pushing the limits of software engineering knowledge on multiple horizons. Successful solutions are not confined to the software architecture and algorithms that comprise an application. Approaches to data architectures and deployment platforms are indelibly intertwined with the software design, and all these dimensions must be considered together in order to meet system scalability requirements. This chapter has described some of the basic principles that underpin system design at scale, and it is the hope of this work that it will spark new research that builds upon the body of software engineering. Scale really does change everything,[32] and as an engineering profession this is a journey that has only just begun.

REFERENCES

[1] P.J. Sadalage, M. Fowler, NoSQL Distilled, Addison-Wesley Professional, 2012.

[29] http://www.appdynamics.com/.
[30] http://en.wikipedia.org/wiki/Nagios.
[31] http://en.wikipedia.org/wiki/Ganglia_%28software%29.
[32] http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=20942.

[2] M. Nygard, Release It!, Pragmatic Bookshelf, 2007.

[3] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, 3rd edition, Addison-Wesley, 2013.

[4] Mark D. Hill, What is scalability?, Comput. Archit. News 18 (4) (December 1990) 18–21, http://dx.doi.org/10.1145/121973.121975.

[5] Leticia Duboc, David S. Rosenblum, Tony Wicks, A framework for modelling and analysis of software systems scalability, in: Proceedings of the 28th International Conference on Software Engineering (ICSE '06), ACM, New York, NY, USA, 2006, pp. 949–952.

[6] A.B. Bondi, Characteristics of scalability and their impact on performance, in: Proc. Second Int'l Workshop on Software and Performance, ACM Press, 2000, pp. 195–203.

[7] G. Brataas, P. Hughes, Exploring architectural scalability, in: Proc. Fourth Int'l Workshop on Software and Performance, ACM Press, 2004, pp. 125–129.

[8] L. Eeckhout, H. Vandierendonck, K. De Bosschere, Quantifying the impact of input data sets on program behavior and its applications, J. Instr.-Level Parallelism 5 (2003).

[9] D.B. Gustavson, The many dimensions of scalability, in: COMPCON, 1994, pp. 60–63.

[10] Gene M. Amdahl, Validity of the single processor approach to achieving large-scale computing capabilities (pdf), in: AFIPS Conference Proceedings, vol. 30, 1967, pp. 483–485.

[11] John L. Gustafson, Reevaluating Amdahl's law, Commun. ACM 31 (5) (May 1988) 532–533, http://dx.doi.org/10.1145/42411.42415.

[12] Lawrence Snyder, Type architectures, shared memory, and the corollary of modest potential, Annu. Rev. Comput. Sci. 1 (1986) 289–317.

[13] Yanpei Chen, Sara Alspaugh, Randy Katz, Interactive analytical processing in big data systems: a cross-industry study of MapReduce workloads, Proc. VLDB Endow. 5 (12) (August 2012) 1802–1813, http://dx.doi.org/10.14778/2367502.2367519.

[14] Y. Kwon, K. Ren, M. Balazinska, B. Howe, Managing skew in hadoop, IEEE Data Eng. Bull. 36 (1) (2013) 24–33.

[15] Len Bass, Ingo Weber, Liming Zhu, Devops: A Software Architect's Perspective, Addison-Wesley, 2015.

[16] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, D. Lewin, Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web, in: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, ACM Press, New York, NY, USA, 1997, pp. 654–663.

[17] Donald E. Knuth, Structured programming with go to statements, ACM Comput. Surv. 6 (4) (December 1974) 261–301, http://dx.doi.org/10.1145/356635.356640.

[18] Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans, Stephen Tu, The HipHop compiler for PHP, SIGPLAN Not. 47 (10) (October 2012) 575–586, http://dx.doi.org/10.1145/2398857.2384658.

[19] Dan Wang, Zhu Han, Sublinear Algorithms for Big Data Applications, Springer-Verlag, 2015.

[20] Jim Webber, Savas Parastatidis, Ian Robinson, REST in Practice: Hypermedia and Systems Architecture, O'Reilly Media, 2010.

[21] John Klein, Ian Gorton, Model-driven observability for big data storage, in: Procs WICSA 2016, Venice, Italy, IEEE, April 2016.

[22] M. Brambilla, J. Cabot, M. Wimmer, Model-Driven Software Engineering in Practice, Morgan & Claypool, 2012.

[23] M.W. Maier, Architecting principles for systems-of-systems, Syst. Eng. 1 (1998) 267–284.

[24] Peter H. Feiler, Kevin Sullivan, Kurt C. Wallnau, Richard P. Gabriel, John B. Goodenough, Richard C. Linger, Thomas A. Longstaff, Rick Kazman, Mark H. Klein, Linda M. Northrop, Douglas Schmidt, Ultra-Large-Scale Systems: The Software Challenge of the Future, Software Engineering Institute, ISBN 0-9786956-0-7, 2006.

[25] http://en.wikipedia.org/wiki/Comparison_of_network_monitoring_systems.

[26] Martin L. Abbott, Michael T. Fisher, The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise, Pearson Education, 2009.