Building Scalable Distributed Systems

Ian Gorton

ISBN:

CHAPTER 1

Introduction to Scalable Systems

The last 20 years have seen unprecedented growth in the size, complexity and capacity of software systems. This rate of growth is hardly likely to slow down in the next 20 years – what these future systems will look like is close to unimaginable right now. The one thing we can guarantee is that more and more software systems will need to be built with constant growth - more requests, more data, more analysis - as a primary design driver.

Scalable is the term used in software engineering to describe software systems that can accommodate growth. In this chapter we will explore what precisely is meant by the ability to scale – known, not surprisingly, as scalability. We'll also describe a few examples that put hard numbers on the capabilities and characteristics of contemporary applications and give a brief history of the origins of the massive systems we routinely build today. Finally, we will describe two general principles for achieving scalability that will recur in various forms throughout the rest of this book and examine the indelible link between scalability and cost.

What is Scalability?

Intuitively, scalability is a pretty straightforward concept. If we ask Wikipedia for a definition, it tells us "scalability is the property of a system to handle a growing amount of work by adding resources to the system". We all know how we scale a highway system – we add more traffic lanes so it can handle a greater number of vehicles. Some of my favorite people know how to scale beer production – they add more capacity in terms of the number and size of brewing vessels, the number of staff to perform and manage the brewing process, and the number of kegs they can fill with tasty fresh brews. Think of any physical system – a transit system, an airport, elevators in a building – and how we increase capacity is pretty obvious.

Unlike physical systems, software is somewhat amorphous. It is not something you can point at, see, touch, feel, and get a sense of how it behaves internally from external observation. It's a digital artifact. At its core, the stream of 1's and 0's that make up executable code and data are hard for anyone to tell apart. So, what does scalability mean in terms of a software system?

Put very simply, and without getting into definition wars, scalability defines a software system's capability to handle growth in some dimension of its operations. Examples of operational dimensions are:

- the number of simultaneous user or external (e.g. sensor) requests a system can process
- the amount of data a system can effectively process and manage
- the value that can be derived from the data a system stores

For example, imagine a major supermarket chain is rapidly opening new stores and increasing the number of self-checkout kiosks in every store. This requires the core supermarket software systems to:

- Handle increased volume from item sale scanning without decreased response time. Instantaneous responses to item scans are necessary to keep customers happy.
- Process and store the greater data volumes generated from increased sales. This data is needed for inventory management, accounting, planning and likely many other functions.
- Derive 'real-time' (e.g. hourly) sales data summaries from each store, region and country and compare to historical trends. This trend data can help highlight unusual events in regions (e.g. unexpected weather conditions, large crowds at events, etc.) and help the stores affected quickly respond.
- Evolve the stock ordering prediction subsystem to be able to correctly anticipate sales (and hence the need for stock reordering) as the number of stores and customers grow

These dimensions are effectively the scalability requirements of a system. If, over a year, the supermarket chain opens 100 new stores and grows sales by 400 times (some of the new stores are big!), then the software system needs to scale to provide the necessary processing capacity to enable the supermarket to operate efficiently. If the systems don't scale, we could lose sales as customers are unhappy. We might hold stock that will not be sold quickly, increasing costs. We might miss opportunities to increase sales by responding to local circumstances with special offerings. All these reduce customer satisfaction and profits. None are good for business.

Successfully scaling is therefore crucial for our imaginary supermarket's business growth, and is in fact the lifeblood of many modern internet applications. But for most business and Government systems, scalability is not a primary quality requirement in the early stages of development and deployment. New features to enhance usability and utility become the drivers of our development cycles. As long as performance is adequate under normal loads, we keep adding user-facing features to enhance the system's business value.

Still, it's not uncommon for systems to evolve into a state where enhanced performance and scalability become a matter of urgency, or even survival. Attractive features and high utility breed success, which brings more requests to handle and more data to manage. This often heralds a tipping point, where design decisions that made sense under light loads are now suddenly technical debt. External trigger events often cause these tipping points – look in the March/April 2020 media at the many reports of Government Unemployment and supermarket online ordering sites crashing under demand caused by the coronavirus pandemic.

Increasing a systems' capacity in some dimension by increasing resources is commonly called *scaling up* or *scaling out* – we'll explore the difference between these later. In addition, unlike physical systems, it is often equally important to be able to *scale down* the capacity of a system to reduce costs. The canonical example of this is Netflix, which has a predictable regional diurnal load that it needs to process. Simply, a lot more people are watching Netflix in any geographical region at 9pm than are at 5am. This enables Netflix to reduce its processing resources during times of lower load. This saves the cost of running the processing nodes that are used in the Amazon cloud, as well as societally worthy things such as reducing data center power consumption. Compare this to a highway. At night when few cars are on the road, we don't retract lanes (except for repairs). The full road capacity is available for the few drivers to go as fast as they like.

There's a lot more to consider about scalability in software systems, but let's come back to these issues after examining the scale of some contemporary software systems circa 2020.

System scale in early 2020's: Examples

Looking ahead in this technology game is always fraught with danger. In 2008 I wrote [1]:

"*While petabyte datasets and gigabit data streams are today's frontiers for data-intensive applications, no doubt 10 years from now we'll fondly reminisce about problems of this scale and be worrying about the difficulties that looming exascale applications are posing.*"

Reasonable sentiments, it is true, but exascale? That's almost commonplace in today's world. Google reported multiple exabytes of Gmail in 2014[1], and by now, do all Google services manage a yottabyte or more? I don't know. I'm not even sure I know what a yottabyte is! Google won't tell us about their storage, but I wouldn't bet against it. Similarly, how much data do Amazon store in the various AWS data stores for their clients. And how many requests does, say, DynamoDB process per second collectively, for all client applications supported. Think about these things for too long and your head will explode.

A great source of information that sometimes gives insights into contemporary operational scales are the major Internet company's technical blogs. There are also Web sites analyzing Internet traffic that are highly illustrative of traffic volumes. Let's take a couple of 'point in time' examples to illustrate a few things we do know today. Bear in mind these will look almost quaint in a year or four.

- Facebook's engineering blog describes Scribe[2], their solution for collecting, aggregating, and delivering petabytes of log data per hour, with low latency and high throughput. Facebook's computing infrastructure comprises millions of machines, each of which generates log files that capture important events relating to system and application health. Processing these log files, for example from a Web server, can give development teams insights into their application's behavior and performance, and support fault finding. Scribe is a custom buffered queuing solution that can transport logs from servers at a rate

[1] https://www.youtube.com/watch?v=eNliOm9NtCM
[2] https://engineering.fb.com/data-infrastructure/scribe/

of several terabytes per second and deliver them to downstream analysis and data warehousing systems. That, my friends, is a lot of data!

- You can see *live* Internet traffic for numerous services at www.internetlivestats.com. Dig around and you'll find statistics like Google handles around 3.5 billion search requests a day, Instagram uploads about 65 million photos per day, and there is something like 1.7 billion web sites. It is a fun site with lots of information to amaze you. Note the data is not really 'live', just estimates based on statistical analyses of multiple data sources.
- In 2016 Google published a paper describing the characteristics of their code base[3]. Amongst the many startling facts reported is: "*The repository contains 86TBs of data, including approximately two billion lines of code in nine million unique source files.*" Remember, this was 2016.

Still, real, concrete data on the scale of the services provided by major Internet sites remain shrouded in commercial-in-confidence secrecy. Luckily, we can get some deep insights into the request and data volumes handled at Internet scale through the annual usage report from one tech company. You can browse their incredibly detailed usage statistics here from 2019 here[4]. It's a fascinating glimpse into the capabilities of massive scale systems. Beware though, this is Pornhub.com. The report is not for the squeamish. Here's one PG-13 illustrative data point – they had 42 billion visits in 2019! I'll let interested readers browse the data in the report to their heart's content. Some of the statistics will definitely make your eyes bulge!

How did we get here? A short history of system growth

I am sure many readers will have trouble believing there was civilized life without Internet search, YouTube and social media. By coincidence, the day I type this sentence is the 15 year anniversary of the first video being uploaded to YouTube[5]. Only 15 years. Yep, it is hard for even me to believe. There's been a lot of wine under the bridge since then. I can't remember how we survived!

So, let's take a brief look back in time at how we arrived at the scale of today's systems. This is from a personal perspective – one which started at college in 1981 when my class of 60 had access to a shared lab of 8 state-of-the-art so-called *microcomputers*[6]. By today's standards, micro they were not.

The 1980s

An age dominated by mainframe and minicomputers. These were basically timeshared multiuser systems where users interacted with the machines via 'dumb' terminals. Personal computers emerged in the early 1980s and developed throughout the decade to become useful business and (relatively) powerful development machines. They were rarely networked however, especially early in the decade. The first limited incarnation of the Internet emerged during this

[3] https://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext

[4] https://www.pornhub.com/insights/2019-year-in-review

[5] https://kogo.iheart.com/content/2020-04-23-youtube-celebrates-15th-anniversary-by-featuring-first-video-ever-posted/

[6] http://pcmuseum.tripod.com/comphis4.html

time[7]. By the end of the 1980s, development labs, universities and increasingly businesses had email and access to exotic internet-based resources such as Usenet discussion forums[8] – think of a relatively primitive and incredibly polite reddit.

1990-1995

Personal computers and networking technology, both LANs and WANS, continued to improve dramatically through this period. This created an environment ripe for the creation of the World Wide Web (WWW) as we know it today. The catalyst was the HTTP/HTML technology that had been pioneered at CERN by Tim Berners-Lee[9] during the 1980s. In 1993 CERN made the WWW technology available on a royalty-free basis. And the rest is history – a platform for information sharing and money-making had been created. By 1995, the number of web sites was tiny, but the seeds of the future were planted with companies like Yahoo! in 1994 and Amazon and eBay in 1995

1996-2000

During this period, the number of web sites grew from around 10,000 to 10 million[10], a truly explosive growth period. Networking bandwidth and access also grew rapidly, with initially dial-up modems for home users (yep, dial-up) and then early broadband technologies becoming available.

This surge in users with Internet access heralded a profound change in how we had to think about building systems. Take for example a retail bank. Before providing online services, it was possible to accurately predict the loads the bank's business systems would experience. You knew how many people worked in the bank and used the internal systems, how many terminals/PCs were connected to the bank's networks, how many ATMs you had to support, and the number and nature of connections to other financial institutions. Armed with this knowledge, we could build systems that support say a maximum of 3000 concurrent users, safe in the knowledge that this number could not be exceeded. Growth would also be relatively slow, and probably most of the time (eg outside business hours) the load would be a lot less than the peak. This made our software design decisions and hardware provisioning a lot easier.

Now imagine our retail bank decides to let all customers have Internet banking access. And the bank has 5 million customers. What is our maximum load now? How will load be dispersed during a business day? When are the peak periods? What happens if we run a limited time promotion to try and sign up new customers? Suddenly our relatively simple and constrained business systems environment is disrupted by the higher average and peak loads and unpredictability you see from Internet-based user populations.

During this period, companies like Amazon, eBay, Google, Yahoo! and the like were pioneering many of the design principles and early versions of advanced technologies for highly scalable systems. They had to, as their request loads and data volumes were growing exponentially.

2000-2006

The late 1990s and early 2000's saw massive investments in, and technological innovations from so called 'dot com' companies, all looking to provide innovative and valuable online businesses. Spending was huge, and not all investments were well targeted. This led to a little event called the 'dot com crash'[11] during 2000/2001. By 2002 the technology landscape was littered with failed investments – anyone remember Pets.Com? Nope. Me neither. About 50% of

[7] https://www.internetsociety.org/internet/history-internet/brief-history-internet/
[8] https://en.wikipedia.org/wiki/Usenet
[9] https://en.wikipedia.org/wiki/History_of_the_World_Wide_Web
[10] https://www.nngroup.com/articles/100-million-websites/
[11] https://en.wikipedia.org/wiki/Dot-com_bubble

dot com's disappeared during this period. Of those that survived, albeit with much lower valuations, many have become the staples we all know and use today.

The number of web sites grew from around 10 to 80 million during this period, and new service and business models emerged. In 2005, YouTube was launched. 2006 saw Facebook become available to the public. In the same year, Amazon Web Services, which had low key beginnings in 2004, relaunched with its S3 and EC2 services. The modern era of Internet-scale computing and cloud-hosted systems was born.

2007-2020 (today)

We now live in a world with nearly 2 billion web sites, of which about 20% are active. There are something like 4 billion Internet users[12]. Huge data centers operated by public cloud operators like AWS, GCP and Azure, along with a myriad of private data centers, for example Twitter's operational infrastructure[13], are scattered around the planet. Clouds host millions of applications, with engineers provisioning and operating their computational and data storage systems using sophisticated cloud management portals. Powerful, feature-rich cloud services make it possible for us to build, deploy and scale our systems literally with a few clicks of a mouse. All you must do is pay your cloud provider bill at the end of the month.

This is the world that this book targets. A world where our applications need to exploit the key principles for building scalable systems and leverage highly scalable infrastructure platforms. Bear in mind, in modern applications, most of the code executed is not written by your organization. It is part of the containers, databases, messaging systems and other components that you compose into your application through API calls and build directives. This makes the selection and use of these components at least as important as the design and development of your own business logic. They are architectural decisions that are not easy to change.

Scalability Basic Design Principles

As we have already discussed, the basic aim of scaling a system is to increase its capacity in some application-specific dimension. A common dimension is increasing the number of requests that a system can process in a given time period. This is known as the system's throughput. Let's use an analogy to explore two basic principles we have available to us for scaling our systems and increasing throughput.

In 1932, one of the world's great icons, the Sydney Harbor Bridge[14], was opened. Now it is a fairly safe assumption that traffic volumes in 2020 are somewhat higher than in 1932. If you have driven over the bridge at peak hour in the last 30 years, then you know that its capacity is exceeded considerably every day. So how do we increase throughput on physical infrastructures such as bridges?

This issue became very prominent in Sydney in the 1980s, when it was realized that the capacity of the harbor crossing had to be increased. The solution was the rather less iconic Sydney Harbor tunnel[15], which essentially follows the same route underneath the harbor. This provides 4 more lanes of traffic, and hence added roughly 1/3rd more capacity to harbor crossings. In not too far

[12] https://www.internetworldstats.com/stats.htm

[13] https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html

[14] https://en.wikipedia.org/wiki/Sydney_Harbour_Bridge

[15] https://en.wikipedia.org/wiki/Sydney_Harbour_Tunnel

away Auckland, their harbor bridge[16] also had a capacity problem as it was built in 1959 with only 4 lanes. In essence, they adopted the same solution as Sydney, namely, to increase capacity. But rather than build a tunnel, they ingeniously doubled the number of lanes by expanding the bridge with the hilariously named 'Nippon Clipons'[17], which widened the bridge on each side. Ask a Kiwi to say 'Nippon Clipons' and you will understand why this is funny.

These examples illustrate the first strategy we have in software systems to increase capacity. We basically replicate the software processing resources to provide more capacity to handle requests and thus increase throughput, as shown in Figure 1. These replicated processing resources are analogous to the lane ways on bridges, providing a mostly independent processing pathway for a stream of arriving requests. Luckily, in cloud-based software systems, replication can be achieved at the click of a mouse, and we can effectively replicate our processing resources thousands of times. We have it a lot easier than bridge builders in that respect.

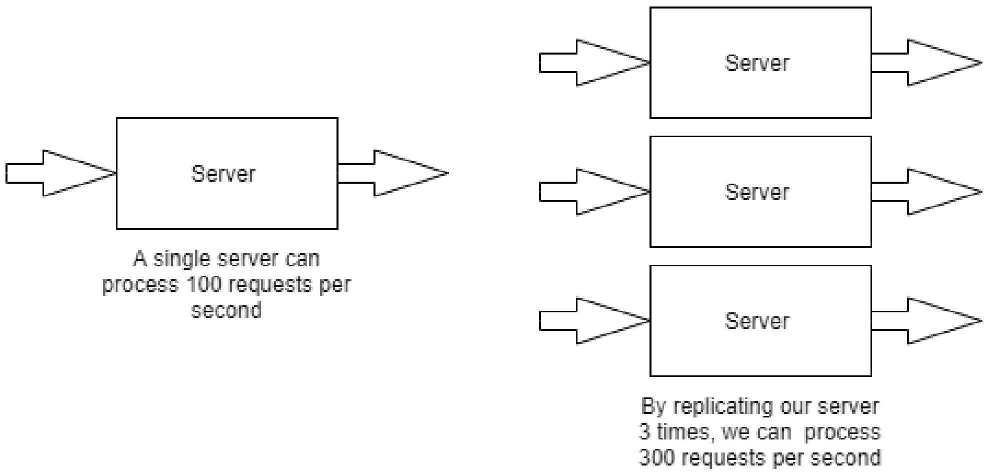
Server
A single server can process 100 requests per second
Server
Server
Server
By replicating our server 3 times, we can process 300 requests per second

Figure 1 Increasing Capacity through Replication

The second strategy for scalability can also be illustrated with our bridge example. In Sydney, some observant person realized that in the mornings a lot more vehicles cross the bridge from north to south, and in the afternoon we see the reverse pattern. A smart solution was therefore devised – allocate more of the lanes to the high demand direction in the morning, and sometime in the afternoon, switch this around. This effectively increased the capacity of the bridge without allocating any new resources – we optimized the resources we already had available.

We can follow this same approach in software to scale our systems. If we can somehow optimize our processing, by maybe using more efficient algorithms, adding extra indexes in our databases to speed up queries, or even rewriting our server in a faster programming language, we can increase our capacity without increasing our resources. The canonical example of this is Facebook's creation of (the now discontinued) HipHop for PHP[18], which increased the speed of Facebook's web page generation by up to 6 times by compiling PHP code to C++.

We'll revisit these two design principles – namely replication and optimization - many times in the remainder of this book. You will see that there are many complex implications of adopting these principles that arise from the fact that we are building distributed systems. Distributed systems have properties that make building scalable systems 'interesting', where interesting in this

[16] https://en.wikipedia.org/wiki/Auckland_Harbour_Bridge
[17] https://en.wikipedia.org/wiki/Auckland_Harbour_Bridge#'Nippon_clip-ons'
[18] https://en.wikipedia.org/wiki/HipHop_for_PHP

context has both positive and negative connotations.

Scalability and Costs

Let's take a trivial hypothetical example to examine the relationship between scalability and costs. Assume we have a Web-based (e.g. web server and database) system that can service a load of 100 concurrent requests with a mean response time of 1 second. 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 load test of this system reveals the performance shown in Figure 2 (left). As the request load increases, we see the mean response time steadily grow to 10 seconds with the projected load. Clearly this is not scalable and cannot satisfy our requirements in its current deployment configuration.

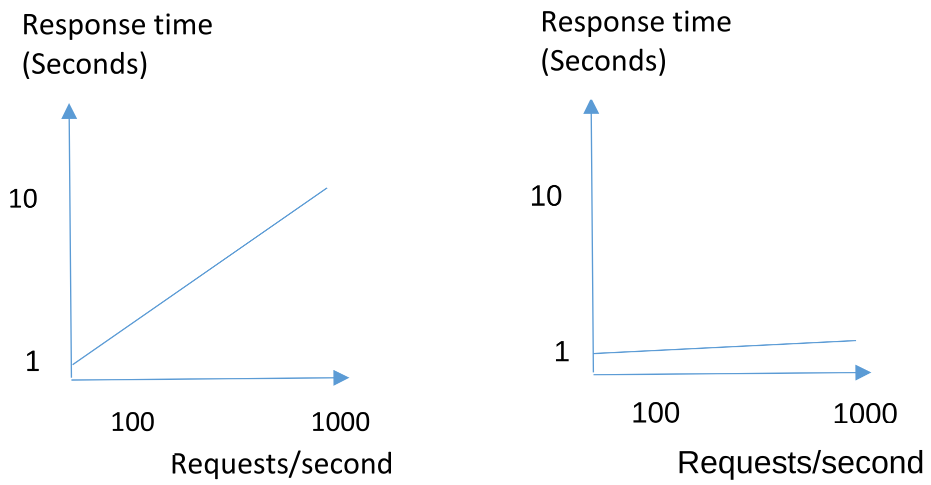

Figure 2 Scaling an application. (Left) – non-scalable performance. (Right) – scalable performance

Clearly some engineering effort is needed in order to achieve the required performance. Figure 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. Party time!

A major question looms however. Namely, how much effort and resources were required to achieve this performance? Perhaps it was simply a case of *scaling up* by running the Web server on a more powerful (virtual) machine. Performing such reprovisioning on a cloud might take 30 minutes at most. Slightly more complex would be reconfiguring the system to *scale out* and run multiple instances of the Web server to increase capacity. Again, this should be a simple, low cost configuration change for the application, with no code changes needed. These would be excellent outcomes.

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

- the database becomes less responsive with 1000 requests per second, requiring an upgrade to a new machine
- 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 more efficiently generate the content,

thus reducing processing time per request.

- 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.
- 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. Another framework? Another programming language even?

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 hours of effort and a thousand dollars extra cloud costs per month for a more powerful server. This is not prohibitively expensive. And let's assume option (4), a rewrite of the Web application layer, requires 10,000 hours 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 hours 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 resource and effort costs are inextricably tied to scalability. If a system is not designed intrinsically to scale, then the downstream costs and resources of increasing its capacity to meet requirements may be massive. For some applications, such as Healthcare.gov[19], these (more than $2 billion) costs are borne and the system is modified to eventually meet business needs. For others, such as Oregon's health care exchange[20], an inability to scale rapidly at low cost can be an expensive ($303million) death knell.

We would never expect 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 technologies to be quickly evolved 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 can more rapidly and cheaply scale up systems to meet rapidly growing demands.

Software systems that can be scaled exponentially while costs grow linearly are known as hyperscale systems, defined as:

"*Hyper scalable systems exhibit exponential growth in computational and storage capabilities while exhibiting linear growth rates in the costs of resources required to build, operate, support and evolve the required software and hardware resources.*"

You can read more about hyperscale systems in this article[21] [3].

[19] https://www.bloomberg.com/news/articles/2014-09-24/obamacare-website-costs-exceed-2-billion-study-finds

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

[21] https://www.researchgate.net/publication/318049054_Chapter_2_Hyperscalability_-_The_Changing_Face_of_Software_Architecture

Summary

The ability to scale an application quickly and cost-effectively should be a defining quality of the software architecture of contemporary Internet-facing applications. We have two basic ways to achieve scalability, namely increasing system capacity, typically through replication, and performance optimization of system components. The rest of this book will delve deeply into how these two basic principles manifest themselves in constructing scalable distributed systems. Get ready for a wild ride.

References

1. Ian Gorton, Paul Greenfield, Alex Szalay, and Roy Williams. 2008. Data-Intensive Computing in the 21st Century. Computer 41, 4 (April 2008), 30–32.
2. Rachel Potvin and Josh Levenberg. 2016. Why Google stores billions of lines of code in a single repository. Commun. ACM 59, 7 (July 2016), 78–87.
3. Ian Gorton (2017). Chapter 2. Hyperscalability – The Changing Face of Software Architecture. 10.1016/B978-0-12-805467-3.00002-8.