Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Joni Väyrynen

Ensuring Availability of a Server-Side Rendered React Application: A Case Study

Master's Thesis
Espoo, April 1, 2019

Supervisors:	Casper Lassenius
Advisor:	Ville Kauppi

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

Author: Joni Väyrynen

Title:
Ensuring Availability of a Server-Side Rendered React Application: A Case Study

Date:	April 1, 2019	**Pages:**	68
Major:	Software and Service Engineering	**Code:**	SCI3043
Supervisors:	Casper Lassenius		
Advisor:	Ville Kauppi		

Service availability is an essential factor for service providers affecting customer satisfaction and the business of the service providers. This thesis presents methods to ensure availability of the service under high load as a case study for Iltalehti, one of the two largest online newspapers in Finland.

The main techniques to ensure availability found in the literature review were the use of load balancing, content delivery networks (CDNs), auto-scaling and improving the performance of server-side rendered React application. Load balancing ensures that all server instances are not affected if one availability zone suffers from failures such as power outage or hardware fails. Content delivery networks lessen the load on the server by caching popular content and therefore improve server performance.

To improve the server performance action research was used where each action research cycle aimed to improve the performance. During each cycle, changes were evaluated by stress testing the test server. The results show that server performance was increased by about a hundred times. Some of the performance increase was achieved by adding more computing power to servers. However, the technical solutions that improved the performance most were client-side rendering fallback, upgrades for Node.js and React versions and reductions of DOM content rendered on the server. Especially, client-side rendering fallback provided great results by improving server performance by slightly over eight times.

Keywords: availability, action research, stress testing, server-side rendering, Node.js, React

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Joni Väyrynen		
Työn nimi: Palvelimella renderöidyn React-sovelluksen saatavuuden varmistaminen: tapaustutkimus			
Päiväys:	1. huhtikuuta 2019	**Sivumäärä:**	68
Pääaine:	Software and Service Engineering	**Koodi:**	SCI3043
Valvojat:	Casper Lassenius		
Ohjaaja:	Ville Kauppi		

Palvelun saatavuus on tärkeä tekijä palveluntarjoajille vaikuttaen asiakkaiden tyytyväisyyteen ja palveluntarjoajien liiketoimintaan. Tämä diplomityö esittelee tapoja varmistaa palvelun saatavuus korkean kuormituksen alla. Työ on tapaustutkimus Iltalehdelle, joka on toinen Suomen kahdesta suurimmasta Internetissä toimivasta uutispalveluista.

Kirjallisuuskatsauksen aikana löydetyt tärkeimmät tekniikat saatavuuden varmistamiseen ovat kuormituksen tasaaminen, sisällönjakeluverkot, automaattinen skaalaus ja palvelimella renderöidyn React-sovelluksen suorituskyvyn parantaminen. Kuormituksen tasaus varmistaa, että ainakin osa palvelininstansseista toimii, vaikka yksi saatavuusalue kärsisi ongelmista, kuten esimerkiksi sähkökatkosta tai laitteistoviasta. Sisällönjakeluverkot vähentävät kuormaa palvelimella ja parantavat palvelinten suorituskykyä tallentamalla suositut sisällöt muistiinsa ja jakamalla niitä suoraan muististaan.

Työssä tehtiin toimintatutkimusta ja jokaisen toimintatutkimussyklin tarkoitus oli parantaa palvelun suorituskykyä. Jokaisen syklin aikana tehdyt muutokset evaluoitiin kuormitustestaamalla testipalvelinta. Työn tulokset osoittavat, että palvelun suorituskyky parani lähes satakertaisesti. Osa suorituskyvyn parannuksesta tuli palvelinten laskentatehon lisäyksestä. Tekniset ratkaisut, jotka paransivat suorituskykyä eniten, olivat selainrenderöintivarasuunnitelma, päivitykset Node.js ja React-versioihin sekä palvelimella renderöitävän sisällön vähentäminen. Erityisesti selainrenderöintivarasuunnitelma toi hyvät tulokset parantamalla suorituskykyä kahdeksan kertaisesti.

Asiasanat:	saatavuus, toimintatutkimus, kuormitustestaus, palvelimella renderöinti, Node.js, React
Kieli:	Englanti

Abbreviations and Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
CDN	Content Delivery Network
CPU	Central Processing Unit
DDoS	Distributed Denial-of-Service
DOM	Document Object Model
HTML	Hyper Text Markup Language
SEO	Search Engine Optimization
SSR	Server Side Rendering

Contents

Chapter 1

Introduction

Failures are inevitable at some point in cloud computing. [8] Failures can be caused by, for example, hardware fails, power outages, software defects, high traffic peaks or distributed denial-of-service attacks. These failures can affect service response times by slowing them or service availability by making service completely inaccessible.

Shortcomings in availability can cause significant harm to users and service providers. Availability issues can prevent users from using the services they need. For service providers, availability problems can cause reputation loss and economic harm. As the business of internet services frequently focus on either users using the money to buy goods or users viewing advertisements, both of those are affected if service is not available.

This thesis presents a case study of ensuring availability on the website of Iltalehti—one of the two biggest online newspapers in Finland. With tens of thousands of concurrent users on the Iltalehti website, a failure in service availability could cost Iltalehti a lot. The primary business model of Iltalehti website is to display advertisements for users and if the site does not work advertisements cannot be shown.

The technology stack of the Iltalehti website was renewed at the end of 2017 and mid of 2018. The new website was made using React JavaScript library. Before the launch of the website, there were shortcomings in server performance which were feared to cause availability issues for the new website. To ensure high availability of the new Iltalehti website, the following research questions are set for this study:

- RQ1: How to ensure availability of the service under high load?
- RQ2: How to improve server-side rendering performance of a React application?

- RQ3: How to auto-scale quickly once high load starts?

The structure for the remainder of this thesis is as follows: Chapter 2 contains a literature review explaining the concept of availability, techniques to improve availability and finally methods to test availability. Chapter 3 presents the case organization Iltalehti, technologies used by case organization and the objectives from the case organization for this thesis. Chapter 4 introduces the research questions and the research method—action research—used for the study. Next, Chapter 5 presents the results of the empirical research in the form of action research cycles. Then, Chapter 6 contains a discussion of the results. Lastly, Chapter 7 concludes the study.

Chapter 2

Background

This chapter takes a look at the concept of availability and techniques to improve it in cloud computing. Section 2.1 defines availability and other concepts closely related to it. Sections 2.2, 2.3 and 2.4 introduce techniques to improve availability in means by cloud computing. Section 2.5 presents means to improve the performance of React application. Lastly, Section 2.6 presents methods to measure availability with load and stress testing.

2.1 Availability

Failures are inevitable at some point in cloud computing. [8] Two dimensions can measure the service failures: extent (how many users or operations are affected by failure) and duration (how long the failure took). If service failure is brief (lasting only a few seconds), users usually retry requesting the resource and availability metrics are not impacted. However, if failure continues for more than a few seconds, users will judge the website being down and probably abandon the service. Therefore, availability metrics are impacted.

Service availability metric can be calculated per uptime and downtime as follows: [8]

$$Availability = \frac{Uptime}{Uptime - Downtime} \tag{2.1}$$

Uptime is not often calculated since it would require constant monitoring. Therefore, another method to calculate availability is to use total service time and downtime:

$$Availability = \frac{TotalInServiceTime - Downtime}{TotalInServiceTime} \tag{2.2}$$

Number of 9's	Service Availability (%)	System Type	Monthly Down Minutes	Practical Meaning (down per year)
1	90	Unmanaged	4,383.00	5 weeks
2	99	Managed	438.30	4 days
3	99.9	Well managed	43.83	9 hours
4	99.99	Fault tolerant	4.38	1 hour
5	99.999	High availability	0.44	5 minutes
6	99.9999	Very high availability	0.04	30 seconds
7	99.99999	Ultra availability	-	3 seconds

Table 2.1: Service Availability and Downtime Ratings [8]

TotalInServiceTime is the time that service is expected to be running during a period, for example, the sum of minutes in a month, and downtime is minutes the service has been unavailable during the same period. For example, using Equation 2.2 with a period of one month (43200 minutes) and downtime 30 minutes, availability would be:

$$Availability = \frac{43200 - 30}{43200} = 99.93\% \tag{2.3}$$

Service availability ratings are often defined by a number of nine's in service availability as seen in Table 2.1. Therefore, in the example Equation 2.3 above, the value of availability has three nine's and would qualify in the well-managed system type. Different services aim for different system type based on the criticality of the service. For example, banks should strive to very high availability rating. On the other hand, services with a large number of users risk losing much more from downtime compared to smaller services due to missing the chance of showing adverts to users, for instance.

Service reliability is a concept very close to service availability. Service reliability represents an ability to provide correct responses in an acceptable time. Whereas availability measures the time that the service has been available, reliability measures portion of the successfully delivered request. Reliability can be calculated similarly to availability as shown in Equation 2.4. [8]

$$Reliability = \frac{SuccessfulResponses}{TotalRequests} \tag{2.4}$$

All failed transactions are not necessarily unsuccessful responses in terms of reliability. [8] For example, a server may return `404 Not Found` when a user navigates to a web page that does not exists and is not even supposed to exist, or `403 Forbidden` when a user tries to visit on a page they are not authorized to view. In these examples failed transactions are successful responses. However, if the server returns responses such as `500 Server Internal Error` or `503 Service Unavailable`, responses are clearly failures and unsuccessful. Also, if the request timeouts (`504 Gateway Timeout`) or the request takes longer than acceptable response time requirement, responses count as failures.

Service response time is an essential factor concerning availability and reliability. Terms response time and latency are often used interchangeably and inconsistently in the literature. In this thesis we will use a definition by Jamae [18]: Latency is the time a request travels across the network. Processing time is the time the server uses processing the request. Thus the time difference between the moment the server received the request and the moment it served to request. Response time is the total time it takes from the moment the user sends the request to the moment they received the response. Therefore, response time is a sum of latency and processing time.

2.2 Content Delivery Networks

Content Delivery Networks (CDNs) improve the quality of service in many ways. CDNs bring an extra layer to client-server communication. With CDN instead of making a request directly to a server, the request is routed first to CDN. If CDN does not have the requested resource in its memory, CDN requests the resource from the server. When CDN receives the resource from the server, it delivers the resource to the client and saves the resource to its memory. Next time the same resource is requested CDN can deliver it straight from its memory. [24]

CDNs improve content delivery speed in two means. Firstly, a typical CDN consists of multiple edge servers distributed around the world. Therefore, there should always be a CDN close to the user. As the origin server is likely located farther away from the user, it takes less time to deliver content from CDN [24]. Secondly, the processing of the resource (HTML for instance) on the origin server may take some time. Therefore, delivering content directly from CDN is faster.

CDNs can also improve the reliability and availability of the service [24]. As popular resources are cached in the memory of CDN, they can be delivered from CDN even if the origin server would be down. In case the origin server is not working, CDN will not help with the new resources, but it can still deliver the popular—although not updating—content such as a front page of the web service.

CDNs also reduce the amount of load on origin servers since content already cached can be delivered directly from CDN. Reduced load on server leads to better availability through server being able to serve more unique requests. Reduced load also means that server capacity may be lowered resulting lower server costs.

While using CDN may lower server costs, the use of CDN itself brings extra costs [24]. In addition to the price, another disadvantage or at least difficulty with CDNs is cache expiration. When serving static non-changing content—such as images—from CDN, caching is not an issue as static content can stay cached practically forever. However, caching becomes more challenging when handling content that may change, such as HTML pages.

The content freshness is especially crucial to media companies. As Dunn and Crosby [14] point out, news articles may receive dozens of modification during the first hours after publication. During that time article HTML cannot stay cached in CDN for too long otherwise readers will not see updates on the article fast enough. On the other hand, news websites may have high traffic peaks due to flash news, such as natural disasters or terrorist attacks, when the use of CDN is necessary to handle the number of increased requests.

CDNs improve availability and reliability, but the protection they offer against distributed denial-of-service (DDoS) attacks is not flawless even with long content cache times. [30] CDNs consist of multiple edge servers and if the requested resources do not exist in the cache of the edge server, the resources have to be requested from the origin server. Therefore, by making requests through all globally distributed edge servers, the attacker may create high traffic on the server. Query strings are another possible attack vector through CDNs. If query strings are allowed, each request to a resource with different query string is forwarded to the origin server.

2.3 Load Balancing

Cloud server outages can have a considerable impact on the availability of cloud systems. Outages may be results from failing software or hardware, power outage or denial of service attacks. [9] As outages may be related to just one location, running more than one server in distributed locations can improve service availability.

Load balancing refers to the process of distributing the workload across the distributed servers. A load balancer serves as a single point of entry for all requests and then forwards them to the instances [10]. If one instance of the distributed system is offline, the load balancer can route all the traffic to the other servers. Therefore, it ensures that the service is all the time available to users. [9]

Load balancing also improves resource utilization and response times by aiming to equalize workload across instances of the system. [2] Load balancing algorithms can be divided into two categories based on whether they take the current system state into account when deciding to which instance request is routed: static and dynamic algorithms. Dynamic load balancing algorithms take the current system state into account. Therefore, they are more probable to provided steadier load to instances and thus better availability.

Load balancing is also the underlying technique enabling the auto-scaling.

2.4 Auto-Scaling

Auto-scaling is a process of changing the server capacity—usually adding or removing computing power or memory—based on need. As explained in the earlier section, a load balancer serves as a single point of entry for all requests and then forwards them to the instances [10]. All instances are running an identical version of the application, and the load balancer aims to keep their load equal. The load balancer has information of all its instances, and when the auto-scaler adds or removes instances, the load balancer will start or stop forwarding requests to them [20].

The main problem with auto-scaling cloud computing environments is to choose the right amount of resources [20]. Lorido-Botran et al. point out that the scaling itself is not difficult; it is difficult to identify how much resources are needed to answer the demand while keeping costs low.

Resource scaling can be divided into two groups—horizontal and vertical. In horizontal scaling, resources are the amounts of instances, and instances are added or removed based on the demand. In vertical scaling resources are the hardware of the instance, for example, CPU power or memory. Based on the need more CPU power or memory is added to or removed from the instance. However, common operating systems do not allow changes on the machine without rebooting them which is why cloud companies usually provide only horizontal scaling. [20]

Each auto-scaler faces problems with provisioning. Provisioning problems can be divided into under-provisioning, over-provisioning, and oscillation. In under-provisioning, the server is not able to handle requests in reasonable time due to a too low amount of resources. On the other hand, in over-provisioning server has more resources than needed to handle the number of current requests. Over-provisioning causes unnecessary costs, especially if the instances are running on low load, but it is better for the end users as it ensures that the service keeps working. Oscillation is a combination of both unwanted events where scaling is done too quickly before seeing the impact of the previous scaling action. Therefore, for example momentarily under-provisioning may lead to over-provisioning. [20]

The four different phases of the auto-scaling process are monitoring, analysis, planning, and execution. [20] To scale correctly, the auto-scaler needs data of system state from a monitoring system. The decisions made by the auto-scaler are based on the performance metrics provided by the monitoring system. Ghanbari et al. [16] presented an extensive list of possible performance metrics divided to seven groups based on what and where is being measured: hardware, general OS process, load balancer, web server, application server, database server, and message queue.

In the next phase, analysis, the metrics collected in the monitoring phase are analyzed. Auto-scalers may either take direct actions based on the analyzed data or make predictions of the future state of the application. Anticipating the future may lead to better results since there is typically a delay in the actions of the auto-scaler. [20]

Once the current state of the system is analyzed or the future predicted, the auto-scaler will plan its actions. The decision made in the planning phase can be to add or remove instances or even to do nothing. Lastly, in the execution phase, the decisions made in the previous phase are being carried out. [20]

Lorido-Botran et al. [20] divide the auto-scaling techniques into five groups based on their underlying theory or technique used to build them: threshold-based rules, reinforcement learning, queuing theory, control theory, time series analysis. In this thesis, we focus only on threshold-based rules as Elastic Beanstalk provides only threshold-based rules for auto-scaling.

In threshold-based rules scaling decisions are made based on the selected performance metric. Also, the scaling action (add or remove instances), the upper and the lower thresholds and durations, and the cooldown period must be defined. [20]. For instance, the selected performance metric could be CPU usage, the scaling action adding or removing two instances, the upper threshold 70% and the lower threshold 30%, both durations two minutes and the cooldown period four minutes. In this case, if CPU usage stays over 70% for over two minutes, two new instances will be added, and then no actions will be taken for the next four minutes.

2.5 React Application Optimization

React is a JavaScript library for building user interfaces developed and maintained by Facebook. Although React was published already in 2015, hardly any literature of the server-side performance of React exists. The main performance issue of server-side rendered (SSR) React is its renderToString function which creates the string presentation of HTML to be sent to a client. The renderToString function is slow due to being synchronous and single-threaded [25].

The most articles of server React improvements [28] [25] offer just simple tips of using the production mode of Node.js and running the minified version of React. Rangel [25] claims that the production mode of Node.js makes React up to 400% faster compared to the development mode as in the development mode React performs lots of error checking which slows React down. According to Rangel, the minified version of React improves performance up to 30%. While these tips promise massive performance improvements, their value is quite low as it could be argued that it is self-evident to use those settings.

However, some articles and blog posts provide more valuable instructions for SSR React. One of those is a blog post of React version 16 [11]. React 16 includes an improved version of the renderToString function. According to Aickin [1] it is 3.8 times faster than the same in React 15 when using Node.js version 8.4 and three times faster when using Node.js version 6. Aickin's performance tests also show that the newer version of Node.js is an excellent performance improvement: in his tests with React 16, server-side rendering with Node.js version 8.4.0 was almost twice as fast as with Node.js version 6.11.3. Arkwright [7] had similar even if not quite as beneficial performance results in his tests: upgrading from Node.js 6 to Node.js 8 while using React 15 yielded 30% performance improvement and upgrading from React 15 to React 16 while using Node.js 8 brought 25% improvement.

Arkwright also presents other interesting techniques to improve the performance of server rendered React applications. The first of those is the use of isomorphic (also known as universal) rendering. While, in traditional React applications all pages are rendered on a client and with server-side rendering on the server, with isomorphic rendering the first page is rendered on the server and following pages on the client. Therefore, with isomorphic rendering, as Arkwright points out, a session including 5 page views causes only one request to the server and the rest of the pages are only rendered on the client. This causes 80% less load for the server compared to traditional server-side rendering. [7]

The second interesting performance improvement presented by Arkwright is a client-side rendering fallback. The idea behind it is to skip the rendering on the server if the load is too high [7]:

1. On the server set up a request queue length counter.
2. When a request is received, increase the value of the counter by one and when a request is served decrease the value of the counter by one.
3. If the queue length is less than the selected value X, proceed as normal.
4. If the queue length is more than the selected value X, skip rendering on the server and let the browser handle rendering based on the data in state management library Redux store.

Arkwright reports around eight times better performance in his benchmarks with the client-side rendering fallback. However, as Arkwright points out, with the client-side rendering fallback all users and search engines will not receive server rendered content and will miss its benefits. On the other hand, eight times better capacity during traffic peaks is a great benefit.

As the rendering components on server is relatively slow, multiple [17] [26] [7] blog posts present techniques to cache rendered component on the server. The idea behind component caching is that most websites have the same components—for example, header, navigation, and footer—repeated on all pages. Therefore, it is a waste of computing resources to re-render them on all requests. Many third-party libraries such as electrode-react-ssr-caching [17] and react-component-caching [26] offer techniques to cache all or selected components and returning those components from the cache instead of rendering them on the following requests.

Grigoryan [17] reports up to 70% performance improvement and Arkwright [7] up to 40% improvement with component caching. The performance gained with component caching is not a static percentage but depends on the number of elements being cached. However, using component caching is not issue free. The third-party component caching libraries rely on the private APIs of React and make changes on them. Therefore, a change in React private API with a new React version may break the component caching of third-party libraries.

2.6 Load, Performance and Stress Testing

To ensure that service works and is available under high load, the service must be tested against a high load. This can be achieved by using load, performance or stress testing.

Load, performance and stress testing are system evaluation techniques where synthetic workload are sent to the system being tested [19]. All the terms are frequently used interchangeably in the literature, but some differences can be found. Load and performance testing aims to simulate a realistic workload. Performance testing focuses on system performance metrics such as response time and availability whereas load testing evaluates how long does it take to complete predefined tasks. The biggest difference in stress testing compared to the other two is that load generated during a stress test puts the system under test at or beyond its capacity. [13]

Load, performance and stress testing for websites are based on a load generator which mimics a user using a browser. The load generator creates virtual users who then send requests to the system under the test. The load generator records each request and can tell the response time or status (successful or failed) for each request. [21] The analysis of system performance and availability can be made based on response times and their statuses. If some requests failed during the test, the service would not have been available to all users under a similar load as during the test. Also, if requests took too long, users might stop waiting and deem that service is not available.

Chapter 3

Case Organization

This chapter takes a look at the case organization Iltalehti. Section 3.1 introduces the company, history of the Iltalehti website and website analytics. Section 3.2 dives deeper into the technology choices and our server architecture. Section 3.3 defines risks which our site may face and which need to be taken into account. Finally, Section 3.4 sets objectives for the thesis for the organization point of view.

3.1 Organization Overview

Iltalehti is a Finnish news and lifestyle media owned by Alma Media Suomi Oy. Iltalehti consists of multiple different topics such as news, sport, entertainment, videos, health, traveling, tech, cars, and fashion. The story of Iltalehti originates back to 1912 and Uusi Suometar newspaper, but regularly Iltalehti tabloid has been published since 1980. Today tabloid newspaper is published on weekdays and Saturdays. The tabloid newspaper and its electronic replica edition had 234,000 daily readers at the end of 2017.

The first Internet version of Iltalehti was published in 1995. At first, online news was only released once per day. Later, the news was also published during significant events and since 2005 news has been published around the clock. In 2013 mobile web version (m.iltalehti.fi) was published alongside the desktop site (www.iltalehti.fi) to provide a better user experience for mobile phone users. Since the start, both mobile and desktop versions of the Iltalehti website had been static websites serving static HTML files. In autumn 2016 we started the project to renew the technology of our website. The new version of the mobile website was published in November 2017. The new website was designed to be responsive, and it was released to desktop users in May 2018 after which the same responsive website was delivered to both mobile and desktop users.

In 2017 Iltalehti.fi was the second biggest Finnish website according to TNS Metrix behind the main competitor Ilta-Sanomat [29]. In 2017 we had the total of 5.4 billion page views (14.9 million per day on average) and 1.4 billion sessions (3.9 million per day on average) across on all our online platforms according to our Google Analytics statistics. In addition to mobile and desktop websites, our online platforms also include native applications to all most important mobile operating systems: Android, iOS, Windows Phone.

In 2017 page views were almost even among the all three platforms the percentages of desktop, mobile and applications being 33%, 32%, and 35% respectively. However, these percentages may vary daily. For example, the portion of mobile page views was 39% on the 18th of August 2017, which was the biggest news day of 2017 with 21.5 million page views. At most, we had 48,278 page views during one minute during that day, which is 804 page views per second on average. 69% of page views came from mobile and desktop sites, so on average, they had 555 page views per second. Unfortunately, Google Analytics only offers page view data in one minute interval, so we do not have exact data of the most page views per second. However, considering some variation, it could be estimated that we had at least 600 page views per second on our websites at some point.

3.2 Technology Overview

Our renewed website is made with React JavaScript library. React is a component-based user interface library which does not make assumptions of other parts of application such as application state management, routing, and API interactions [15]. For state management, we are using Redux and for routing Universal Router which means that our website uses isomorphic rendering. Isomorphic rendering is a combination of traditional client-side rendering and server-side rendering where the first page is rendered on the server and following pages only on the client as explained in Section 2.5.

On the client-side, our website acts as a single page application meaning that after the first page load the site will not make any requests to the server but dynamically changes the content on the page. When a user navigates on the website, the client side JavaScript code determines which API requests the browser has to make to show wanted content to the user. For instance, when a user clicks a link to an article, the client-side code will request the article with specific id from the Iltalehti API and based on the data received from the API, the React code can change the content on the web page and show the article to the user.

On the server-side, we run our React code on Node.js JavaScript runtime. When the server receives a request, it behaves just as a browser making same API requests but instead of rendering content for the browser, the server uses the renderToString function of React. The renderToString function, as the name suggests, creates a string presentation of the HTML, which then can be served for the browser. As the document sent to the browser already contains the HTML presentation of the page, the browser can view content faster for the user compared to client-side rendering where browser first has to load JavaScript code, then make the API request and only then render the content.

In addition to being able to view requested page faster for the user, server-side rendering provides search engine optimization (SEO) benefits. While the biggest search engines, such as Google, can nowadays crawl client-side rendered content, we had issues with only client-side rendered pages with some other engines we were using while developing our new website. The biggest drawback of the server side rendering is its cost as explained later in this thesis.

As discussed in Section 2.5, isomorphic rendering reduces server load since only the first page view generates a request to the server. According to our Google Analytics data, our users viewed 3.55 pages per session in 2017. It can be argued that during each session one request is made to the server—the rest of the page views are processed on the client. Therefore, from the earlier estimated 600 page views per second, only about 170 would cause a request to the server because of the use of isomorphic rendering.

Our website's server infrastructure is run on Amazon Web Services (AWS). For this thesis, the most relevant parts of AWS that we are using are EC2, Elastic Beanstalk, CloudFront and CloudWatch. EC2 (Amazon Elastic Compute Cloud) provides a variety of different type and size cloud computing instances on which servers are run. The instance types vary from general purpose to compute optimized, memory optimized, storage optimized and accelerated computing. During the research for this thesis, we used both general purpose (T2) and computed optimized (C4 and C5) instances. Table 3.2 presents the on-demand prices for the selected T2, C4, and C5 EC2 instances.

Elastic Beanstalk is a combination of AWS services aimed to make deploying and running web application easy. Elastic Beanstalk uses EC2 instances but also offers load balancing, auto-scaling and monitoring by default. [6] The load balancing solution of Elastic Beanstalk works well for us out of the box. We are always running at least two instances on two different availability zones to ensure better availability in case one of the zones becomes

Instance Type	Number of vCPU	Memory (GB)	Price per hour (€)	Price per 30 days (€)
t2.nano	1	0.5	0.0063	4.54
t2.micro	1	1	0.0126	9.07
t2.small	1	2	0.0250	18.00
t2.medium	2	4	0.0500	36.00
t2.large	2	8	0.1008	72.58
t2.xlarge	4	16	0.2016	145.15
t2.2xlarge	8	32	0.4032	290.30
c4.large	2	3.75	0.1130	81.36
c4.xlarge	4	7.5	0.2260	162.72
c4.2xlarge	8	15	0.4530	326.16
c5.large	2	4	0.0960	69.12
c5.xlarge	4	8	0.1920	138.24
c5.2xlarge	8	16	0.3840	276.48

Table 3.1: On demand prices of the selected T2, C4 and C5 EC2 instances at the end of 2018. [5]

unavailable. However, the default auto-scaling solution scales instances only based on network usage, which does not meet our needs as network usage does not always linearly correlate with the load on the server.

In addition to Elastic Beanstalk monitoring, we are also using Amazon CloudWatch as a monitoring service. CloudWatch supports more metrics compared to Elastic Beanstalk monitoring and allows us to send additional metrics as well. In CloudWatch we can draw different kind of charts from selected metrics. We mostly use line charts and Figure 3.1 presents an example of charts we are using. We also use CloudWatch to monitor other services than Elastic Beanstalk, for example, CloudFront. In addition to monitoring, we are using CloudWatch to send alarms if specific terms are fulfilled, for example, if request amount to our servers becomes higher than expected.

We use Amazon CloudFront as our content delivery network (CDN). Depending on the page we cache responses for 10-60 seconds in CloudFront. When the page is cached on the CloudFront, a request for the page does not cause a request to the server as the page is just delivered from the memory of CloudFront. Considering the behavior of our users, most of the requests are delivered from the CDN. In 2017 the front page of our website was the landing page for 61% of our users. Therefore, the front page was almost

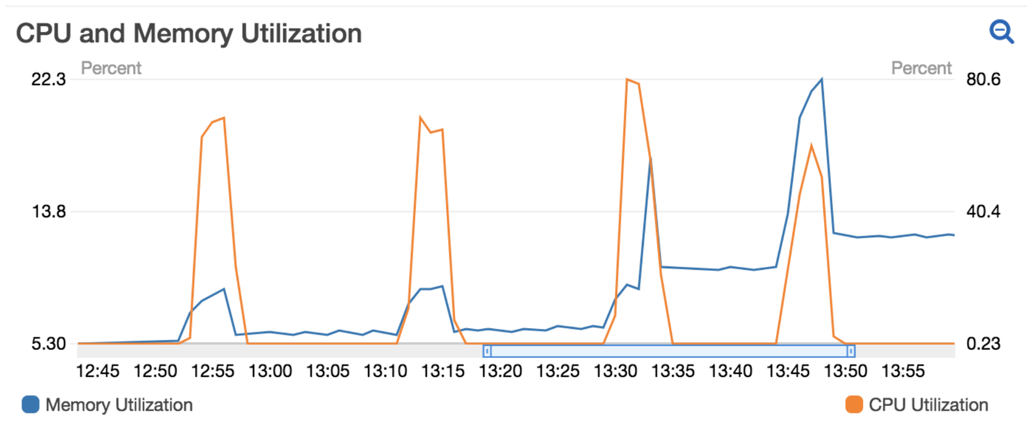


Figure 3.1: An example of a CloudWatch chart plotting memory and CPU utilization during stress tests

always delivered from CDN.

There is a lot more variation for the rest of the landing pages. The second most visited landing page in 2017 was the front page of the news section (iltalehti.fi/uutiset) being the landing page for only 1.7% users. However, when we examine daily data, we see that just a small amount of page views have a unique landing page or cause a request to the server. For example, on 8th of February 2018, an average news day with 14.9 million page views on all platforms, which of 4.9 million on mobile, Google Analytics shows that there were 23,164 different landing pages on the mobile website. Our CloudWatch data shows us that on the same day we had 614,000 requests on the server of the mobile site which means that only 12.5% of page views caused requests to the server on that day. This combined to use of isomorphic rendering reduces the number of requests to the server to a low number. Out of the estimated 600 page views per second it can be estimated that only 20-40 cause a request to the server.

Figure 3.2 summarizes the server architecture of the Iltalehti website and the data flow when a user requests a page, for example, an article. The request goes first to CloudFront, and if the HTML of the requested article is in the cache of CloudFront, it is delivered directly from there. Otherwise, CloudFront requests the page from the server. The request is then routed to the load balancer which requests the page from a selected instance. Within the instance, the request is directed to Nginx process running on a chosen CPU core. Nginx works as a proxy, and in case of an article page, it routes the request to Node.js process running the React application. The React application requests the article data from the Iltalehti API and once all needed data is received React application creates the HTML of the page which is then routed through CloudFront to the user.

The Iltalehti API plays a crucial role in the availability of Iltalehti website. If the API does not work, then also the site will not be able to display any content. However, in this thesis, we focus on the other aspects affecting the availability of the Iltalehti website and expect the API to work correctly. Nonetheless, many findings of this thesis also apply to the Iltalehti API as well.

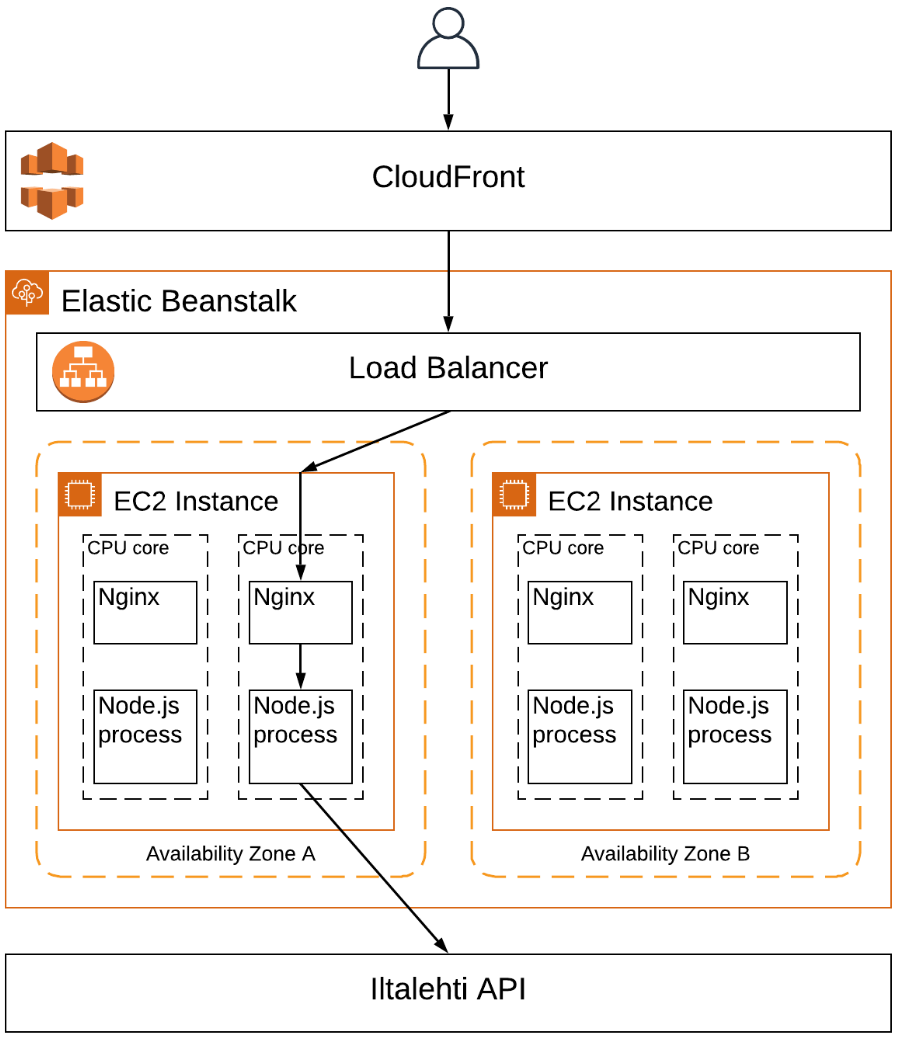


Figure 3.2: Summary of the server architecture of the Iltalehti website

3.3 Risk Analysis

Considering the earlier analytic statistics, in a normal situation, we can serve a high amount of pages with a very low load on the server side. However, we also have to have some capacity on the server to deal with the situation where something goes wrong. We can identify three possible threats to our server which are a failure in the client-side React JavaScript code, a failure in CloudFront cache and a distributed denial-of-service (DDoS) attack.

A failure in the client-side React code could mean that the JavaScript code does not get executed at all or breaks at some point which means that all page changes would cause a request to the server. This failure could multiply the number of requests to the server by two or three, but as long as CloudFront cache works, the impact to the server performance would be quite low. The more significant impact would be on client-side user experience as all features would not work, and to the financial side, as we would not be able to display advertisements. The probability to JavaScript error is quite low as the error should go unnoticed through all our development instances. Even if the error went to the production, we would be able to revert to the earlier error-free version within a few minutes.

At worst a failure in CloudFront could mean that our whole website would not work. Our domain name system points to CloudFront address, and if CloudFront would not work and retrieve data from our server, the user would not get their requested content, and the website would not work. Another possible failure in CloudFront, which would have an impact on our servers, could be an error caching requests and retrieving them all from the server. Considering the earlier estimation of how much CloudFront reduces requests to the server, this could make server load almost ten times higher.

CloudFront protects against DDoS attacks in certain situations. For example, if an attacker makes requests to a single URL, most of the requests are returned from CloudFront. Since we are not caching per query strings, adding them does not help the attacker. However, when an object expires from the cache, CloudFront lets all request through to the server until the server responses with the new object which CloudFront then caches and serves for following requests. CloudFront also does not help if the attacker keeps making requests to new URLs.

3.4 Objectives

Our primary objective is to have as close as possible to 100% uptime on our website with reasonable costs. While 100% availability might usually be impossible to achieve, availability should at least be closer to 99.99%. If the costs were not a limitation, we would be able to accomplish that by simply using an almost infinite amount of server capacity and not to take concern for anything else. However, as we want to do that as economically as possible, we need to take into account other aspects as well.

Considering the risks explained in the previous section, we concluded that our server must be able to serve at least 1,000 requests per second at any time. With that performance, our server would remain responsive during the biggest news day of the year with over 600 requests per second even though both client-side rendering and CloudFront stopped working as expected.

Performance of 1000 requests per second is also a decent buffer against a DDoS attack even if in most cases it will not be enough. Therefore, our server performance must be able to scale based on traffic. However, selecting the right amount of resources to scale is hard. After a discussion, we decided to set an objective that our server must be able to serve 3000 requests per second—three times more than in the normal state—within 5 minutes from the start of the high request rate.

To achieve these objectives, we have to first optimize our server performance so that the server can handle as much traffic as possible. Secondly, we need to use auto-scaling to adapt to the changing amount of traffic and not to run an excessive amount of servers close to the idle state when the amount of traffic is low.

Chapter 4

Methods

This chapter presents the chosen research method for the study. The main objective of this study is to ensure the availability of service under high load. Section 4.1 introduces research questions. Section 4.2 presents used research method: action research.

4.1 Research Questions

The main objective of this study is to ensure the availability of the Iltalehti website under the high load. The other objectives we set in Section 3.4 were that at any time our server must be able to serve at least 1000 requests per second and that within five minutes of the start of high load the server must be able to serve 3000 requests per second, thus trebling its performance. Based on these objectives we formulated the following three research questions:

- RQ1: How to ensure availability of the service under high load?
- RQ2: How to improve server-side rendering performance of a React application?
- RQ3: How to auto-scale quickly once high load starts?

RQ1 aims to find answers to our primary objective and is already discussed to some extent in Chapters 2 and 3. RQ2 and RQ3 can be considered as subquestions of RQ1 as their answers increase the performance of the system and thus also the availability. RQ2 and RQ3 also search for answers to our other two objectives.

4.2 Action Research

From the start, it was clear to us that our research would be an iterative and cyclical process where each step taken would aim to increase the performance of our server and thus its availability under the high load. Therefore, we chose to use action research.

Action research cycles are defined in multiple different ways in the literature. For example, Susman and Evered [27] present five phases: diagnosing, action planning, action taking, evaluating and specifying learning. We, however, chose to follow the model of the action research cycle by Coghlan and Brannick [12] as presented in Figure 4.1.

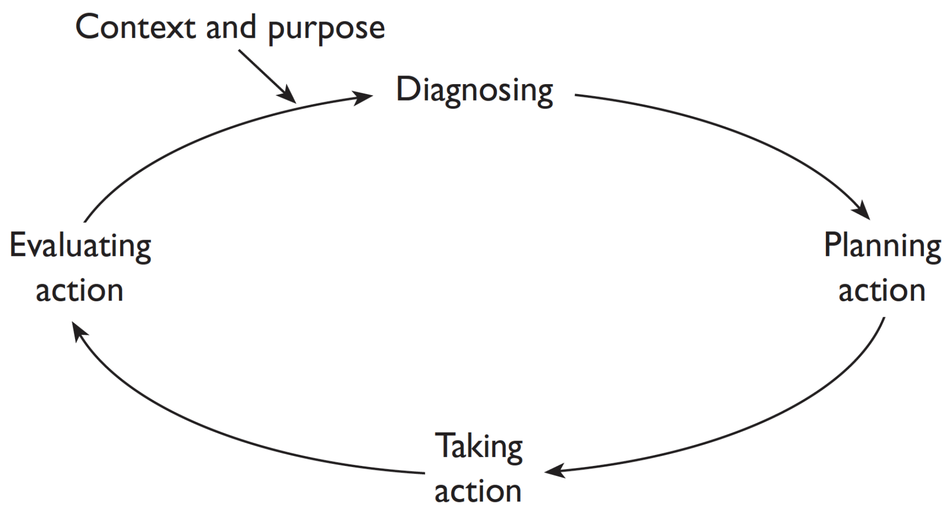


Figure 4.1: The action research cycle by Coghlan and Brannick [12]

The only small change we did to Coghlan's and Brannick's model was combining planning and taking action in our results reporting. Our research included both steps, but they are reported as one to reduce repetition since frequently taking action would have just included words: "Execute actions as planned." Therefore, the action research cycle steps presented in this research are:

1. Diagnosing
2. Planning and taking action
3. Evaluating

The following subsections describe the above steps in our research.

4.2.1 Diagnosing

The main issue through our research was that our server was not able to serve enough requests per second. The main reason for this was that the performance of the server side rendering of React was too low. The main diagnosis did not change during the research as there was not much we could do to change the server-side rendering function of React.

However, on each cycle, we also tried to find other smaller issues and fix those to improve the overall performance. On some cycles diagnosis came from the results of the previous cycle where the actions made did not yield the expected results.

4.2.2 Planning and Taking Action

The planning of action for each cycle was made either based on the smaller issues found during diagnosing or on the performance improvement ideas invented by us or found in the literature. During the planning, we also made assumptions of how changes should improve the performance.

The acting included making the planned changes to the code or the server settings. Once the changes had been made and tested locally, they were deployed to an EC2 instance running on Elastic Beanstalk and then evaluated.

4.2.3 Evaluation

Evaluation of actions was done by stress testing the test server. We used the Gatling Load Testing tool to conduct our stress tests. We chose Gatling as we had already used it to stress test our other products. We installed Gatling to the EC2 instance from which we ran the load to the test server deployed on taking action step.

In Gatling, each stress test is a simulation which consists of three main parts: HTTP configuration, scenario definition, and user injection. The URL of the target website with the desired request headers are defined in the HTTP configuration. The wanted HTTP method (we only used the GET method), and the path or paths of the pages under the test are specified in the scenario definition.

We used a scenario with a list of different paths which were requested in order repeatedly. The list contained paths for 10000 different articles (our article path format was at the time `/section/article_id`, for example `/kotimaa/201802042200718691`). Even though we ran the load straight to Elastic Beanstalk (we bypassed CloudFront) we wanted to simulate a situation where the server gets a large number of different requests. In production, if the same article is requested repeatedly, it will be returned from the cache of CloudFront.

The virtual users who make the actual requests are generated in the user injection part of the Gatling simulation. As we wanted to run stress tests with different amount of requests per second, we injected a large number of users to the simulation and then throttled them. In throttling, we defined the number of requests per second, how many seconds it should take for the simulation to reach the selected amount of requests and for how many minutes we wanted to hold desired request amount. Unless otherwise stated, we kept ramp-up time in 30 seconds and ran each test for 3 to 5 minutes. Our Gatling simulation code is presented in Appendix A.

Until the auto-scaling tests, each stress test was run against a single instance. We used only one instance to lessen the amount of load needed to bring the server to the edge of its capacity and to make monitoring of server status easier. Also for monitoring purposes, we kept the requests per second rate for each stress test constant. We could have done a test scenario where the requested amount kept steadily increasing but finding the performance limit of the server would have been harder and the results less reliable.

We got results from the stress tests from two sources: from the Gatling simulation report and by monitoring the CPU and memory usage of the server. Once Gatling has run the simulation, it provides an HTML document presenting the results of the stress test. An example of simulation results is presented in Appendix B. At the top of the results page is a bar chart showing a number of requests divided to four bars: requests below 800 milliseconds (green bar), requests between 800 and 1200 milliseconds (yellow bar), requests over 1200 milliseconds (orange bar) and lastly failed requests (red bar). This bar chart shows quickly whether simulation was successful or not. If all requests were below 800 milliseconds, the system was performing correctly. If most requests were below 800 milliseconds but some between 800 and 1200, the system is running close to its limit. Further, if there were a significant number of requests above 1200 milliseconds or some even failed, it indicates that there was too much load for the system to handle.

Following the bar chart, the result page has two tables. The first table, statistics, presents response statuses and response time percentiles for each requested paths. As we are making requests to up to 10000 different pages, this information is not very usable. However, the top line of the table has response time percentiles for all requests and shows, for example, that 50% of requests were delivered in over 5 seconds. The top line gives us more information on how long requests took compared to the earlier bar chart which only tells that requests took more than 1200 milliseconds. The second table shows the error codes and their counts and has little use for us since we already see from the bar chart if some requests failed.

There are more charts below the tables. The line chart of active users along the simulation provides not useful information for us. The next chart, a bar chart of response time distribution, gives us similar information as the response time percentiles for all requests from the statistics table but just from a slightly different angle. Below that is an area chart of response time percentiles over time during the simulation. This chart tells us whether the responses took a long time during the whole simulation or was there just one peak during which response times were long.

The last two charts present the number of requests and the number of responses per second. If the simulation went well, the number of requests per second was just straight lines after the first 5 seconds (the time to reach the desired requests per second rate). However, it seemed that if the load was way over the capacity of the server, Gatling started to throttle the number of requests it made. On the other hand, responses per second line usually had some small variation (plus or minus 10%) compared to requests per second, especially if the server was running on high load.

All in all, our analysis made based on the results of Gatling followed the following pattern: if all requests were below 800 milliseconds, the server was able to handle that load, and therefore we added more requests per second and ran another stress test. If we saw some requests between 800 and 1200 milliseconds, we knew that the system was running close to its limit. We still added some more requests to see if responses went then over 1200 milliseconds. Further, if response times were mostly over 1200 milliseconds, we started to analyze what was the time percentile for the worst half of requests. Also, we inspected the response time percentiles chart to see if there was just a short period during the simulation when results were poor.

With Gatling, we were able to find the limits of the current system setup, but we did not get any information of why the system reached to its limit or when the limit was not reached, how close to the limit the system was. To analyze each stress test further, we needed information on systems CPU and memory utilization. We monitored those with two different ways.

During the first stress tests, we connected to the instance via ssh and monitored the CPU and memory utilization using the Linux Top command. The problem with the Top command is that it only displays the current CPU and memory usage. Therefore, we constantly had to keep watching their statuses during the simulations. However, we at least got some information on the CPU and memory statuses even if the method for obtaining the data was not ideal.

We wanted instances automatically log CPU and memory usage data to be able to analyze their behavior during the simulation. We found out that Beanstalk has enhanced health reporting option which allows us to log additional metrics to CloudWatch. Supported metrics include application latency percentiles, number of requests (total and by response code) and CPU usage with different states: CPUIrq, CPUUser, CPUIdle, CPUSystem, CPUSoftirq, CPUIowait, CPUNice. While the other states only use up to a few percents of CPU time, we monitored CPUUser since that told us how much processing power our code was using. Therefore, if the CPUUser value was around 95%, it meant that the CPU was under full load.

The enhanced health reporting of Elastic Beanstalk provided us data of CPU usage but not memory usage which Beanstalk does not offer out of the box for some reason. However, AWS provides separate CloudWatch monitoring scripts that allow users to log memory utilization of EC2 instance to CloudWatch [3]. So, when creating a new instance, we also had to install those monitoring scripts to get memory logs of the instances.

CloudWatch presents us the data collected from the instances as line charts. By plotting the CPU and memory utilization data to the same chart, we were able to conclude their correlation and by repeating stress tests to see how the system recovered from the earlier load. Unfortunately, CloudWatch only collects the average CPU and memory usage data once a minute, so we were not able to compare the CPU and memory usage data to for example response time percentiles over time chart of Gatling. However, it provided us valuable data, for example, to explain anomalies in repeated stress tests.

Chapter 5

Results

This chapter presents the results of our research. A summary of the action research cycle steps taken during each cycle are presented at the end of each section. Sections 5.1-5.9 (Cycles 1-9) aim to answer RQ2 whereas Section 5.10 (Cycle 10) looks answers for RQ3. Finally, Section 5.11 summarizes the results from Cycles 1-9.

5.1 Cycle 1: Initial Performance

The first step was to evaluate the initial amount of requests our website was able to serve per second. At this point, we were running our server on an EC2 t2.micro instance, which is the second cheapest and the least powerful instance of all EC2 instances. The t2.micro instance has only one virtual CPU and just 1GB of memory. We never even planned to run our servers on an as weak instance in the production but that was the instance type we used during the development.

Nevertheless, we did not expect as bad results as what we got. We started with a low amount of 10 requests per second and as Figure 5.1 shows about 90% of requests took more than 1.2 seconds. Furthermore, half of the requests took more than 4.5 seconds.

Ten requests per second were indeed way below our needs, but we still wanted to find the initial limit of our system. Next, we tested with only five requests per second, and the server was able to handle that load reasonably well serving 99% of requests below or at 455 milliseconds. Still, even with five requests per second CPU utilization was over 80%. With eight requests per second couple of percent of requests took more than 800 milliseconds. However, with nine requests per second for only 40% of requests response time stayed below 800 milliseconds. Therefore, the initial limit of the server

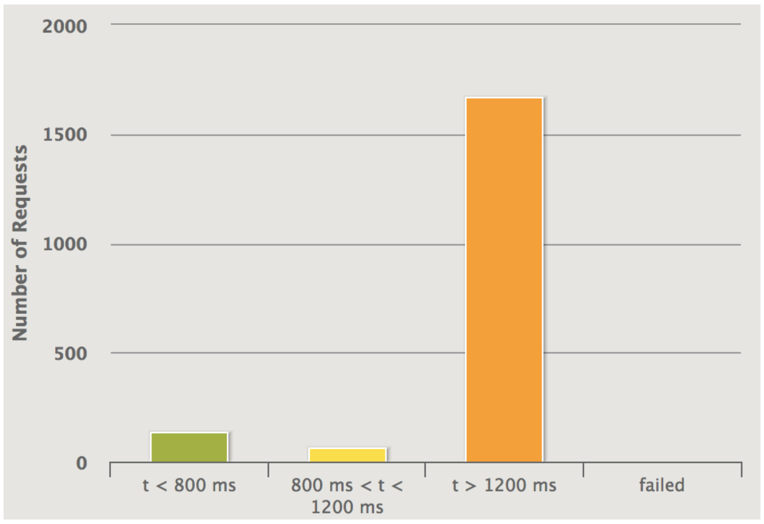


Figure 5.1: Response time distributions for the first stress test of 10 requests per second

was eight requests per second which meant that we would need 125 t2.micro instances to achieve the desired performance of 1000 requests per second. Running 125 t2.micro instances would cost 1133 euros per month which is more than we wished to pay.

5.2 Cycle 2: More Computing Power

The first problem we were able to identify was obviously the use of weak instance type. While upgrading the instance type would not decrease the costs, since doubling the computing powers also doubles the costs, it would make the server more manageable by running less amount of instances. Therefore, we upgraded the instance type from t2.micro to t2.medium, which has two virtual CPUs and 4GB of memory—twice and four times more respectively compared to the t2.micro instance type.

Considering that we doubled the amount of virtual CPUs, we also expected to double the performance. However, that was not the case as Figure 5.2 displaying response times shows. The t2.micro instance in the previous test set was able to serve eight requests per second, but the t2.medium instance with double amount of virtual CPUs was not able to serve 15 requests per second with reasonable response time. Almost 95% of requests took more than 1.2 seconds and 50% even more than 26 seconds.

Decreasing request amount to 12 per second did not change much other

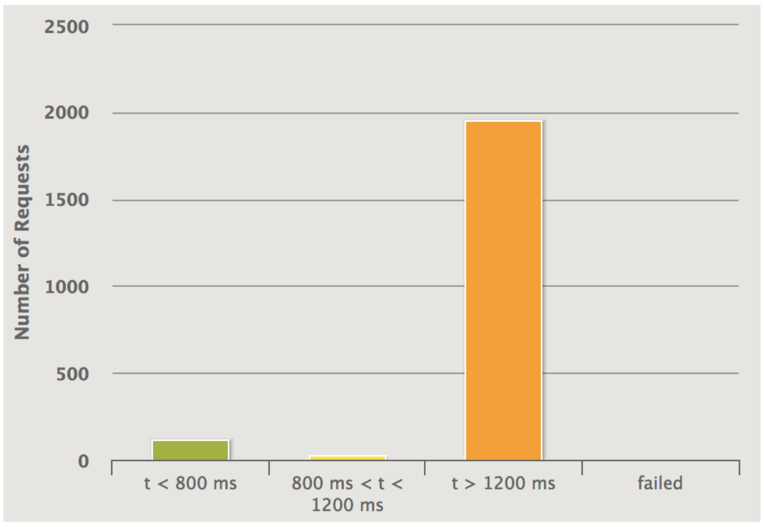


Figure 5.2: Response time distributions for the t2.medium instance with 15 requests per second

than now 50% of requests took more than 9.5 seconds. On the other hand, with ten requests per second, almost all requests were served below 800 milliseconds. So, by doubling the CPU power and having four times more memory, the server was able to serve only two more requests per second being only 25% increase when we were expecting 100% increase.

Table 5.1 summarizes the steps taken in this cycle.

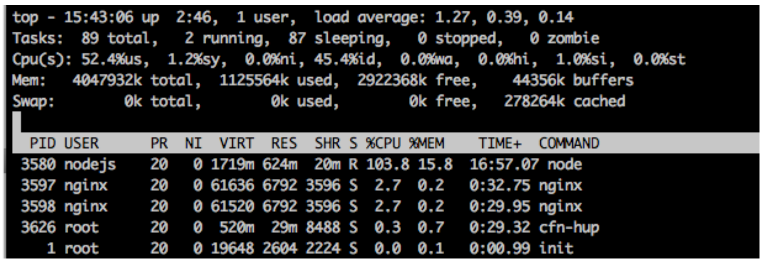
Cycle Step	**Action**
Diagnosing	The t2.micro instance is the second weakest instance type of all EC2 instances.
Planning and taking action	Upgrade instance type from t2.micro to t2.medium which has two CPUs instead of one. This should double the server performance.
Evaluating	The server performance was only increased by 25% instead of expected 100% increase.

Table 5.1: The action research cycle steps for Cycle 2

5.3 Cycle 3: Node.js Clusters

There were some problems within our system as doubling the CPU capacity did not double the performance, and we wanted to find a reason for that. We repeated the previous stress test with 15 requests per second while monitoring server health status from the health overview of Elastic Beanstalk. The health overview showed us that during the test the maximum CPU utilization was only about 53%. Therefore, almost half of the CPU capacity was not even in use.

The Linux Top command showed similar results on the server compared to the health overview. The total CPU utilization was slightly over 50% during the test as seen in Figure 5.3 (52.4% at the time of the screenshot). Interestingly, there was only one Node.js process running at 103.8% of the capacity of the one CPU core. Therefore, it seemed like the server was able to run only one Node.js process which could only use the capacity of one CPU core.



```
top - 15:43:06 up  2:46,  1 user,  load average: 1.27, 0.39, 0.14
Tasks:  89 total,   2 running,  87 sleeping,   0 stopped,   0 zombie
Cpu(s): 52.4%us,  1.2%sy,  0.0%ni, 45.4%id,  0.0%wa,  0.0%hi,  1.0%si,  0.0%st
Mem:   4047932k total,  1125564k used,  2922368k free,    44356k buffers
Swap:        0k total,        0k used,        0k free,   278264k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 3580 nodejs    20   0 1719m 624m  20m R 103.8 15.8  16:57.07 node
 3597 nginx     20   0 61636 6792 3596 S  2.7  0.2   0:32.75 nginx
 3598 nginx     20   0 61520 6792 3596 S  2.7  0.2   0:29.95 nginx
 3626 root      20   0  520m  29m 8488 S  0.3  0.7   0:29.32 cfn-hup
    1 root      20   0 19648 2604 2224 S  0.0  0.1   0:00.99 init
```

Figure 5.3: The Top command on the server during a stress test

It turned out that by default Node.js can use only one CPU core [23]. To be able to divide the load to multiple CPU cores, one must start a cluster of Node.js processes. Listing 5.1 presents an example of how we used the Node.js cluster module. When the server starts, the cluster code creates a new Node.js process until there are as many processes as there are CPU cores in the system. For example, in the case of the t2.medium instance, which has two CPU cores, the cluster module creates two processes.

```
// Other server imports:
// ...
import cluster from 'cluster';
import os from 'os';
const numCPUs = os.cpus().length;

if (cluster.isMaster) {
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
} else {
  // Server code here
  // ...
}
```

Listing 5.1: The usage of the Node.js cluster module

We implemented the cluster module to our server code and moved on to evaluate changes by running more stress tests. As the system running on the t2.medium instance had managed to serve ten requests per second with only one Node.js process, we first tested the cluster of processes with 20 requests per second.

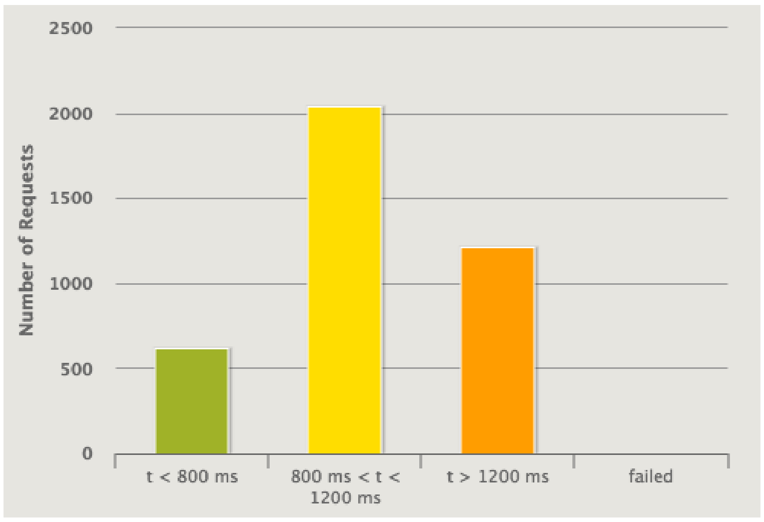


Figure 5.4: Response times and status for the first cluster test with 20 requests per second

However, as Figure 5.4 shows the server was able to serve only 15% of requests below 800 milliseconds. It is understandable that the server could not double its performance even with two CPU cores being able to run Node.js processes. Although the first CPU core was running Node.js process on full load, the other processor was not idle. The other CPU core still had to run the system and Nginx, for example. Now, with two cores running Node.js, both cores still had to run the system and two Nginx processes and not to dedicate all the computing power to Node.js alone.

We next tested with 15 requests per second, and the server managed that smoothly. With 17 requests per second, about 5% of requests took more than 800 milliseconds. With 18 and 19 requests per second for 25% and 40% of requests, response time was more than 800 milliseconds. Therefore, the limit of the t2.medium instance running with as many Node.js processes as there were CPU cores was 16-17 requests per seconds, which doubles the performance compared to the t2.micro instance.

We also wanted to test what would be the optimal amount of Node.js process compared to the number of CPU cores. Therefore, we deployed a version running with the double amount of Node.js processes compared to the number of CPU cores—4 processes in the case of t2.medium instance. First, we repeated the earlier 17 requests per second test, and the server managed to serve all requests below 800 milliseconds. With 19 requests per second, 13% of requests took more than 800 milliseconds, so the limit of the system was 18-19 requests per second.

As the limit the server was able to serve was two requests higher with the instance running with double amount of Node.js processes compared to the number of CPU cores, we decided that doubling the amount of process provided better performance and did the following tests with that setup. However, our conclusion of the best amount of Node.js processes was wrong as explained later in Section 5.7.

Table 5.2 summarizes the steps taken in this cycle.

5.4 Cycle 4: Reducing the Amount of DOM Nodes

Until this point, we had not improved the performance of the application itself but only increased the amount of computing power. As explained earlier in Section 2.5 issues with server-side rendering of React come from the renderToString function which is relatively slow at processing content. For that reason, as also discussed in Section 2.5, some have developed libraries that cache rendered components on the server.

Based on this we came up with an idea of reducing content rendered on the server. By giving renderToString function less content to render we expected it to run faster. Therefore, the server should be able to serve more requests

Cycle Step	**Action**
Diagnosing	Doubling the amount of CPUs did not double the performance as by default Node.js process runs only on a single CPU core.
Planning and taking action	Implement Node.js cluster module to spawn a cluster of processes each of which can be assigned to its own CPU core. Test with an equal amount of clusters as CPUs and double amount of cluster compared to CPUs.
Evaluating	With two clusters (same amount as CPUs) the server performance was now doubled compared to an instance with only one CPU. Four clusters provided slightly better performance the server being able to serve two requests more than a server running only two clusters.

Table 5.2: The action research cycle steps for Cycle 3

per second. Earlier, we had been rendering precisely the same content on both server-side and client-side. This meant rendering plenty of content that the users did not see until they started to scroll on the page. Therefore, we were causing unnecessary load for the server by rendering content that could have been only rendered on the client.

We selected five random articles and calculated the amount of DOM (Document Object Model) nodes rendered on server by running the following code in console of the browser with JavaScript disabled:

```
document.getElementsByTagName('*').length;
```

We disabled JavaScript because we wanted only to count the number of nodes rendered on the server and not additional DOM content generated by advertisements, for instance. The average amount of DOM nodes in the selected five articles was 670.

Next, we reduced the amount of server-side rendered content. We shortened the list of latest articles on the right column from 40 articles to 15, removed social media content, and removed the list of most read articles from the bottom of the article. The average amount of the DOM nodes for the same five articles was 362 after the content reduction. Therefore, the amount of DOM nodes was reduced by 46%.

We repeated the earlier stress test, and the server had no issues to serve 20, 25 and 30 requests per second. However, 35 and 32 requests per second put response times for the most requests above 1.2 seconds. The limit of the server with reduced DOM content was 30 requests per second. Therefore, by reducing the amount of DOM nodes by 46%, we were able to increase the limit of the server from 19 requests per second to 30 which was 58% improvement.

Table 5.3 summarizes the steps taken in this cycle.

Cycle Step	**Action**
Diagnosing	Server side rendering is slow.
Planning and taking action	Reduce the amount of DOM content by shortening article lists and removing the content on the bottom of articles on the server. This should make server-side rendering faster.
Evaluating	By reducing the content rendered on the server by 46% increased the server performance by 58%.

Table 5.3: The action research cycle steps for Cycle 4

5.5 Cycle 5: Upgrading to React 16

On September 26th, 2017 Facebook released React 16 which promised about three times better performance for server-side rendering compared to React 15 while using Node.js 6. [11] However, they pointed out in the blog post that performance upgrade might not be the same in real-world systems compared to their benchmarks. By this point, we had been using Node.js 6 with React 15.

In the end, upgrading to React 16 provided a relatively small improvement for the server performance. We ran the first stress test with 35 requests per second, and 2% of requests took more than 800 milliseconds. That was the limit of the server as with 37 requests per second 85% of requests took more than 1.2 seconds and 5% even failed. Therefore, React 16 increased the performance by only 17% compared to the earlier limit of 30 requests per second.

Table 5.4 summarizes the steps taken in this cycle.

Cycle Step	Action
Diagnosing	Server side rendering is slow.
Planning and taking action	Upgrade to React version 16 which reportedly could make server side rendering performance up to three times better.
Evaluating	React 16 improved performance just by 17%.

Table 5.4: The action research cycle steps for Cycle 5

5.6 Cycle 6: Fixing Memory Problems and Upgrading to Node.js Version 8

Since the performance of 35 requests per second was still way below our objectives, we next upgraded our instance type to c4.xlarge. It has four virtual CPUs and 7.5 GB of memory compared to 2 virtual CPUs and 4 GB of memory on the previously used t2.medium instance. In addition to having more CPUs, the processors of the C4 instance family are more advanced than the ones on the T2 instance family.

During the stress tests of the new instance type we noticed inconsistencies in the stress test results. When we ran the same test repeatedly without restarting the server, the results became increasingly worse. When we ran the same stress test (40 requests per second) five times in a row, we received some alarming results as seen in Figure 5.5.

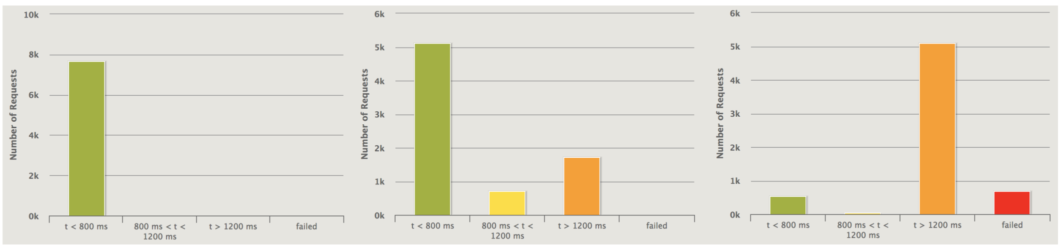


Figure 5.5: Response times and status for the first test (left), the third test (middle) and the fifth test (right) with each 40 requests per second

The first two tests gave good results where all response times were under 800 milliseconds. However, on the third test around 10% requests took 800-1200 milliseconds and 20% over 1.2 seconds. Results got even worse on the following attempts as on the fifth stress test about 80% of requests took more than 1.2 seconds (most of them took actually over 20 seconds) and even 11%

of requests failed. Therefore, after the fifth test, we would have concluded that the server was not able to handle the load even if after the first stress test conclusion would have been opposite.

By connecting to the server with SSH, we were able to detect that the memory utilization was relatively high after the stress tests even if there was no load on the server. Because monitoring memory and CPU usage on the server did not work very well, we started logging those to CloudWatch as described in Section 4.2.3. Once the CloudWatch logs were enabled, we rebooted the server and repeated the previous five stress tests.

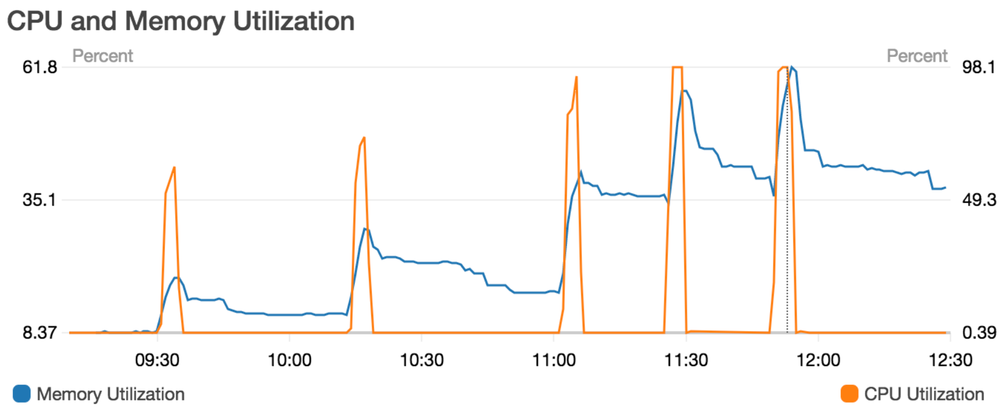


Figure 5.6: CPU and memory utilization during and after five stress tests with each 40 requests per second

As seen in Figure 5.6 the memory usage did not recover to the previous levels after each test. During the load memory utilization increased as expected, but once the load on the server stopped, memory utilization decreased only slightly. Given some more time, about 5 minutes for the first stress test and about 20 minutes for the second, memory utilization decreased some more, but it was still higher than before starting each test.

Still, memory usage never reached 100%. At highest it was 61.8% during the fifth stress test. Therefore, the reason for poor performance was not in the running out of memory. For some reason, as seen in Figure 5.6, the CPU usage was higher, the more memory was used. During the first test memory usage peaked at 20% and CPU usage was 61%. On the second test memory usage increased to 29% and CPU usage to 72%. On the third test memory usage peaked at 41% and CPU usage was 94%. During the third test, we started to see significantly slower response times. On the last two test CPU usage was 100% on both times and memory usages were 56 and 61%.

So far, we had been running our server using version 6.11.1 of Node.js.

We had already updated our React version to 16 earlier and React blog [11] promised "about a 3x performance improvement in Node.js 6" compared to React 15. The blog also said that you would get "a full 3.8x improvement in the new Node.js 8.4 release". Therefore, we repeated the tests running the server with Node.js 8.4.0.

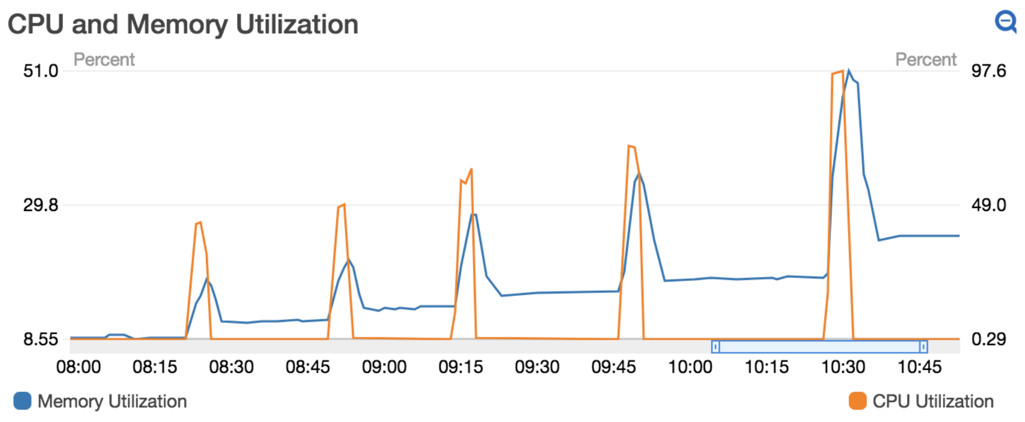


Figure 5.7: CPU and memory utilization during and after five stress tests with server running Node.js 8.4.0

Using Node.js 8.4.0 improved the performance as Figure 5.7 shows. For the first four stress tests, memory usage increased only by few percents compared to the memory usage before the stress test. This result was a significant improvement compared to the tests run on Node.js 6.11.1 where each test increased the memory usage by around 10%. Still, even with the upgraded Node.js version, memory usage kept increasing after each test, and during the fifth test it peaked at 51%, and CPU was on full load.

A week after our Node.js 8.4.0 tests, AWS started providing Beanstalk platform which supported new Node.js version 8.8.1. We repeated once again the same five stress tests as earlier with the latest Node.js version, and the results were significantly better than before as Figure 5.8 presents.

After the same five tests as before (40 requests per second) memory usage increased from 8.2% before tests to 9.5% after all tests. Also, during the tests memory usage was a lot lower compared to the same tests that were run on Node.js 8.4.0. With Node.js 8.4.0 memory usage peaked at 17% even during the first test and during the fifth test it increased up to 51%. With Node.js 8.8.1 the highest amount of memory being used was 13.2% during the fifth test while during the first test memory usage was about 1% lower. Due to significantly lower memory consumption, CPU usage also stays much lower being at most 41% under the load.

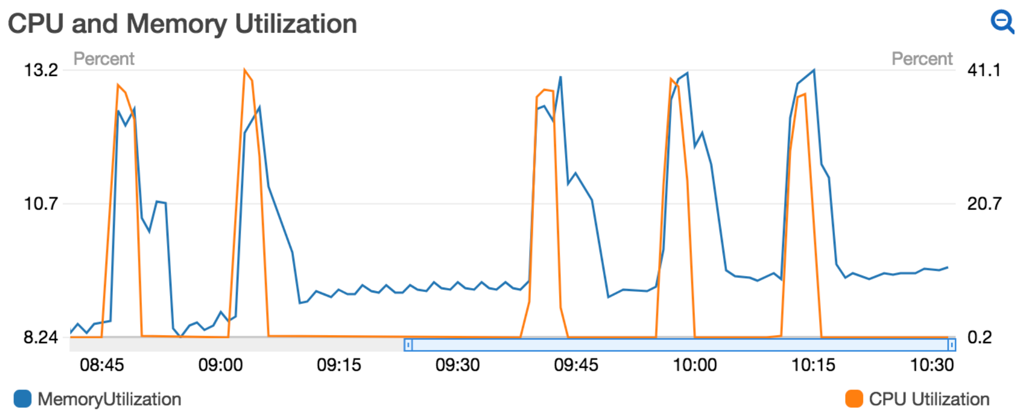


Figure 5.8: CPU and memory utilization during and after five stress tests with server running Node.js 8.8.1

With the huge performance increase with Node.js 8.8.1, the next step was to find new request limit the c4.xlarge instance type was able to serve. We used the same stress test profile as before (ramp up to the wanted amount of requests in 30 seconds and hold that for three minutes) but increased the number of requests.

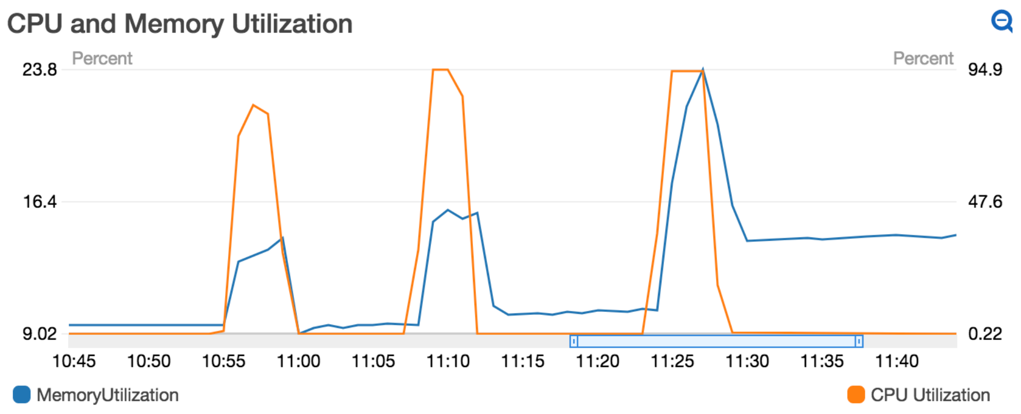


Figure 5.9: CPU and memory utilization with 70, 80 and 90 requests per second

The server handled 70 requests per second without any issues and response times stayed below 800 milliseconds while CPU usage was about 80% as seen in Figure 5.9. However, with 80 requests per second, the server started to hit the limit. Nearly half of the requested took more than 1.2 seconds although all requests were still served under 4 seconds. CPU started to hit max load during the test.

Therefore, it comes with no surprise that 90 requests per second were much more than the server could serve in a reasonable time. Almost 90% of requests took more than 1.2 seconds. Furthermore, 50% of requests took even more than 12 seconds and 2% of requests also failed. What was even more alarming was that memory usage did not recover to the level before the test. After the first two tests (70 and 80 requests per second) memory usage was only about 1% higher than before the tests being at 10%. After the third test (90 requests per second) memory usage stayed at around 14% and did not get any lower over time. So even with Node.js 8.8.1 we still had some memory problems when the server was under higher load than it was able to handle.

Table 5.5 summarizes the steps taken in this cycle.

Cycle Step	**Action**
Diagnosing	Memory usage does not revert back to previous levels after each stress test. The higher the memory usage the higher CPU usage and the worse server performance.
Planning and taking action	Upgrading Node.js to version 8 should provide better performance compared to Node.js 6. Upgrade first to Node.js 8.4.0 and then to Node.js 8.8.1.
Evaluating	With Node.js 8.4.0 memory usage increment after each stress test was lesser than with Node.js 6. With Node.js 8.8.1 memory consumption did not increase almost at all.

Table 5.5: The action research cycle steps for Cycle 6

5.7 Cycle 7: Another Node.js Cluster Test

All stress tests after the earlier Node.js cluster test done in Cycle 3 had been run with twice as many Node.js processes as there where CPU cores. However, considering the memory issues with Node.js 6.11.1, the earlier tests might have been biased due to higher memory usage than expected. Therefore, we repeated the stress test with 80 requests per second twice with still running double amount of Node.js process compared to CPU cores. Then we reduced the amount of Node.js processes to equal the number of CPU cores and repeated the same test.

As the Figure 5.10 shows, there was an improvement in both CPU and memory utilization when running an equal amount of clusters compared to CPUs. Whereas memory consumption was 15% during the first two tests, it was only about 10% during the third and fourth tests. This also reflected CPU usage which stayed at around 90% instead of 100% and furthermore to response times which all stayed below 800 milliseconds.

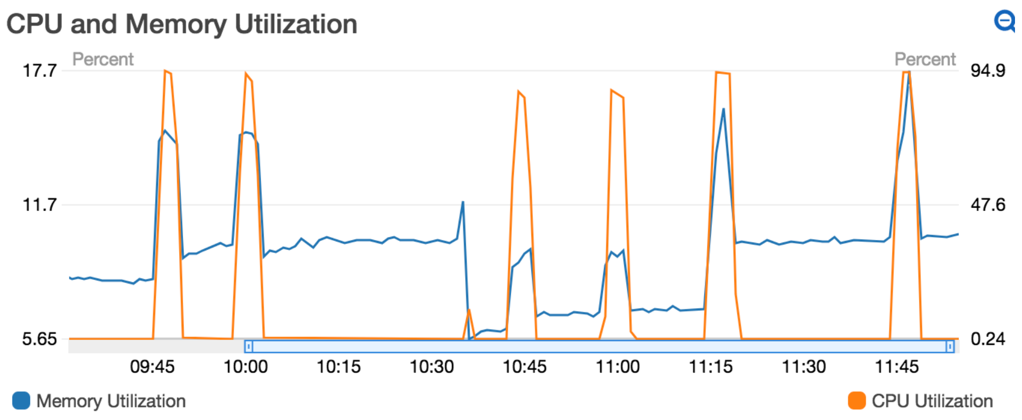


Figure 5.10: 80 requests with twice as many clusters compared to CPUs (peaks 1 and 2); 80 requests with equal amount of clusters compared to CPUs (peaks 3 and 4); and 90 requests with equal amount of clusters compared to CPUs (peaks 5 and 6)

We also run tests with 90 requests per second to the server running the same amount of Node.js processes as CPU cores (the fifth and sixth tests in Figure 5.10). This load was again too much for the server to handle, but the results were still slightly better compared to the previous test on the server running double amount Node.js processes. None of the responses failed, and 50% of requests were served in below 7.5 seconds compared to 12 seconds with the double amount of Node.js processes. All in all, running the same amount of Node.js processes as CPU cores gave better results compared to running a double amount of processes.

Table 5.6 summarizes the steps taken in this cycle.

5.8 Cycle 8: Upgrading the Instance Type

In November 2017 AWS published a new C5 instance family [4]. Reportedly, C5 instances had better processors than C4 instances and included more memory (8GB instead of 7.5 GB for the xlarge model, for example). In

Cycle Step	Action
Diagnosing	The earlier Node.js cluster test might have been biased due to memory issues.
Planning and taking action	Stress test two servers one running equal amount of clusters compared to CPUs and other running double amount of clusters.
Evaluating	On the contrary to the earlier Node.js cluster stress test, the equal amount of clusters provided better results.

Table 5.6: The action research cycle steps for Cycle 7

addition to better computing power, the C5 instances cost about 17% less than the C4 instances. Considering better computing power combined with a lower price, we were eager to test the new instance type.

We changed the instance type from c4.xlarge to c5.xlarge and ran two stress tests with 80 requests per second. As Figure 5.11 shows the CPU utilization was at most at 69% during the tests (the first two peaks of Figure 5.11). That is considerably lower compared to the same test run on c4.xlarge instance where CPU utilization was at 88%. During the next two tests with 90 requests per second CPU utilization was at most at 80% but still, the server was not able to serve all requests.

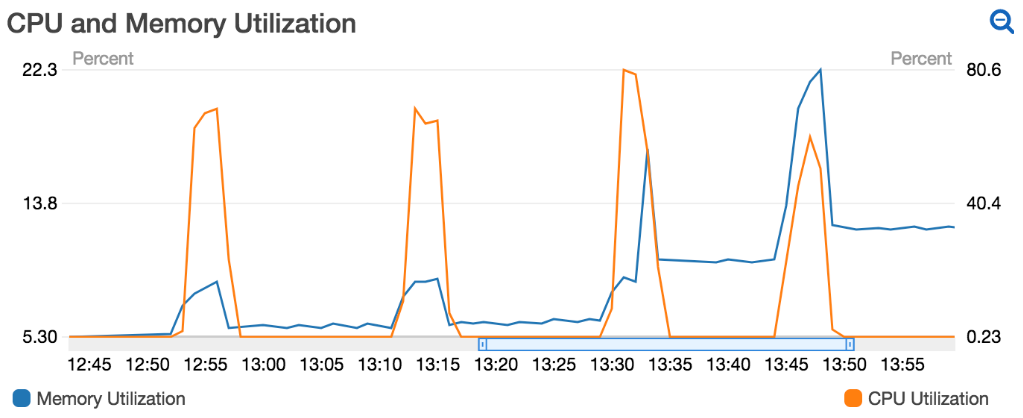


Figure 5.11: CPU and memory utilization for c5.xlarge instance (80, 80, 90 and 90 requests per second)

This behavior was strange as we had expected that CPU power was our only limiter since we also had no issues with memory usage. We went through our system logs and found many following warnings from the Nginx error log:

```
1024 worker_connections are not enough
```

It turned out we had to increase the amount of worker_connections in the Nginx config to allow more connections and also increase the ulimit values of Linux system [22]. With the increased values, the server was able to serve 90 requests per second with 15% requests taking over 800 milliseconds. With 95 requests per second, 6% of requests took more than 1.2 seconds. Therefore, the limit of a single c5.xlarge instance was slightly over 90 requests per second which is around 10% more compared to c4.xlarge.

Table 5.7 summarizes the steps taken in this cycle.

Cycle Step	**Action**
Diagnosing	The new c5.xlarge instance was not able to response to all requests even if CPU usage was at most 80% due to too low Nginx worker_connections and system ulimit values.
Planning and taking action	Increase the values of Nginx worker_connections and system ulimit.
Evaluating	With increased values c5.xlarge instance was able to serve slightly over 90 requests per second within acceptable response times.

Table 5.7: The action research cycle steps for Cycle 8

5.9 Cycle 9: Client-Side Rendering Fallback

Nearly 100 requests per second per instance were already a decent amount of requests, but we still wanted to improve that. To reach our objective of 1000 requests per second with the current performance we would need 11 instances and the costs would be high. One exciting way to increase the number of requests a single instance can handle was client-side rendering fallback presented by Arkwright [7]. As explained in Section 2.5, if the number of requests were high on the server, the server would skip rendering and rendering of the page would only happen in the browser. To make this happen, the server needs to keep the count of request it has not yet served. Therefore, whenever a request comes to the server, the server needs to increase request queue length counter by one and when server responses, decrease the counter by one. Then, if the request queue length counter is above the selected limit, the server skips the content rendering part.

We took Arkwright's idea even a bit further. Arkwright's setup included fetching data on the server and passing it as the initial Redux state in HTML, but we skipped that as well. If the counter was above the limit, we only rendered HTML containing the React JavaScript bundle script tags, few other necessary scripts, and the style sheets. Also, we saved that HTML to memory and for the following requests we were able to serve the HTML string straight from memory, which further increased performance of the system.

We tested different request queue limits and ended up to 15. As a c5.xlarge instance has four virtual CPUs that resulted in a total of 60 requests in the queue before we skipped server-side rendering (SSR). We ran tests with a constant load for five minutes increasing the number of requests for each test and monitoring CPU usage and the number of requests rendered on the server. With 60 requests per second, all requests were rendered on the server as expected. With 70 requests per second, SSR was skipped for one percent of requests and the average CPU usage was 85%. With 80 requests per second, SSR was skipped for 12% of requests and average CPU usage was 95%. All in all, by setting the total request queue limit to 60 requests resulted in being able to server-side render about 70 requests per second.

Next, we tested how many requests server was able to handle in total with the client-side rendering fallback. We started with 100 requests per second and already during that test CPU was running on full load. For 7% of requests, response time was over 1.2 seconds and the maximum response time was slightly below 4 seconds. Earlier, without the client-side rendering fallback, we would have judged that as an unacceptable result and increasing the number of requests per second would have provided increasingly worse results. However, this time as we kept increasing the amount of requests response times increased only slightly as Table 5.9 presents.

Up to the 800 requests per second at most 10% of requests took more than 1.2 seconds, which, considering the significant request increment, is an excellent result. With 900 requests per second response times started rising and with 1000 requests per second for 70% of requests responses from server took over 1.2 seconds and 2% of requests even failed. Therefore, the acceptable limit for one c5.xlarge instance was about 800 requests per second with the client-side rendering fallback. That is about eight times more compared to the earlier state without the client-side rendering fallback and a similar result as in Arkwright's [7] tests.

Considering that a single c5.xlarge instance can handle up to 800 requests per second in reasonable time, two c5.xlarge instances can handle together up to 1600 requests which is already clearly past our objective of 1000 requests

Requests per second	Server-side rendered requests (%)	Requests with response time over 1200 ms (%)
100	71	7
200	34	10
300	22	10
400	16	7
500	13	7
600	10	6
700	9	6
800	8	8
900	7	19
1000	6	70

Table 5.8: SSR request percetages and latencies

per second. Running two c5.xlarge instances costs 276 euros per month which is four times less than what running the initial system—being able to run 1000 requests per second—would have cost. By running three c5.large instances instead of two c5.xlarge instances would have made the costs cheaper while still reaching our objective but we are happy to pay little extra to be able to serve more server-side rendered content.

Table 5.9 summarizes the steps taken in this cycle.

Cycle Step	**Action**
Diagnosing	Server-side performance is not high enough.
Planning and taking action	Skip rendering on the server if server is receiving a large number of requests.
Evaluating	Client-side rendering fallback made server able to serve 8 times more requests per second.

Table 5.9: The action research cycle steps for Cycle 9

5.10 Cycle 10: Auto-Scaling

After achieving our first objective of being able to serve 1000 requests per second, the next step was to reach our next goal of being able to serve three times more requests within five minutes of the start of high load. Although we would only need to double the instance count from two c5.xlarge instances to four c5.xlarge instances to reach the performance of 3000 requests per second, we wanted auto-scaling to triple the count of instances.

We considered three metrics from the auto-scaling trigger options of Elastic Beanstalk: latency, request count and CPU utilization. The other options—network in/out, disk write/read or healthy/unhealthy host count—were not suitable metrics for our use case. From the three that we considered, we first dismissed the latency metric. While increasing latency usually indicates that the system is not working correctly under the load, that was not the case due to our use of client-side rendering fallback. Principally, we would want to serve each request with the complete HTML content. If we scaled based on latency, the scaling action might not occur as the average latency might stay low even while serving hundreds of requests per second as when the server skips rendering, the response is served from memory which happens quickly.

After dismissing the latency option, we tested auto-scaling with the request count metric. However, whichever scaling option combination we selected, we could not get auto-scaling work with request count as a trigger option. We tried it with different units of measurement (count and count/second), different trigger statistics (average and sum), very low upper scaling thresholds and ran a high load to the server but Elastic Beanstalk still would not scale the instance count.

Therefore, we ended up using CPU utilization as our scaling metric. For the statistic, we chose the average CPU utilization instead of the maximum as we did not want to scale if the CPU usage peaked just for a second within consecutive minutes. The CPU unit we measured was a percentage and the initial upper and lower thresholds 75% and 35% respectively. The scale-up increment and scale down decrease were both set to one instance. The period between metric evaluations was set to one minute which is the lowest possible value in Beanstalk. Since the goal for scaling was to happen as quickly as possible, the breach duration was set to two minutes.

To focus on scaling, we used c5.large instances to lessen the number of requests needed to keep server over 75% threshold. We ran a stress test with 300 requests per second and as the Figure 5.12 presents it took 15 minutes to increase the number of instances from two to six once the load on the server started. This result was way below our goal to triple the server capacity within five minutes of the time when the load started.

There were two clear options of how to speed up our scaling: decrease the time between scaling iterations or scale up more than one instance at the time. With our initial scaling setup scaling iteration took about three minutes—two minutes for the scaling cooldown and the two metrics evalu-

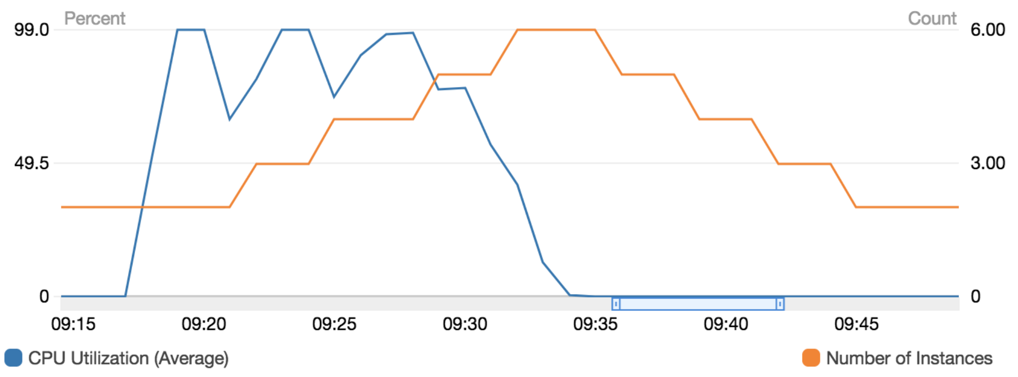


Figure 5.12: CPU utilization and number of instances during the first auto-scaling test

ations and then about 40 seconds to add a new instance. We considered decreasing the number of evaluations before scaling triggers from two to one but decided against it, as if increased traffic lasted under one minute, scaling would be unnecessary.

Therefore, we tested scaling with the scale-up increment of two instances instead of one. As Figure 5.13 shows, it took 7 minutes to scale to six running instances.

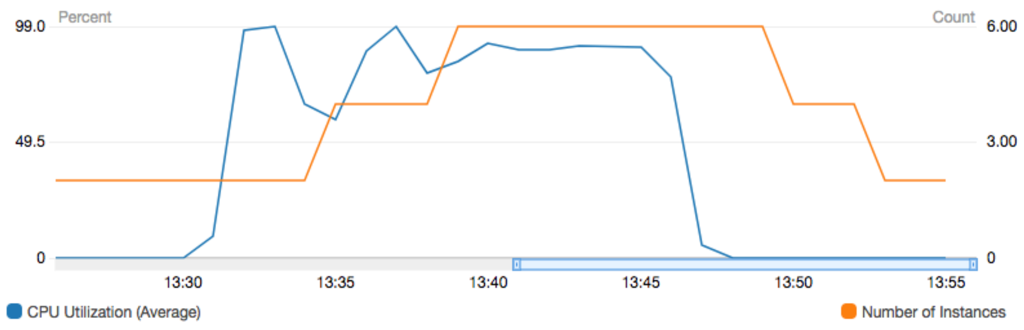


Figure 5.13: CPU utilization and number of instances with scaling increment of two instances

Even though scaling cooldown was set to just one minute, the second scaling took one minute more than the first one. The reason for this, as Figure 5.13 shows, is that during the scale up the average CPU usage dropped below 75% because while booting up new instances their CPU usage stayed low. As we could not get the scale up increment of two instances to triple instance count within five minutes, the only option left was to scale up with four instances as presented in Figure 5.14.

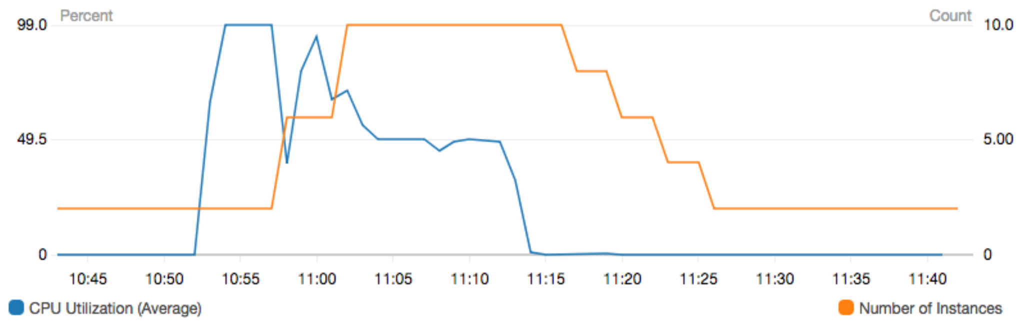


Figure 5.14: CPU utilization and number of instances with scaling increment of four instances and scale down decrease of two instances

For some reason, this time the first scale up took four minutes which still meets our objective. After another four minutes, the auto-scaler added another four instances meaning that within eight minutes we had ten instances running. Also, as can be seen in Figure 5.14, even though we set the scale-up increment to four instances, we kept the scale down decrease in two instances. We figured that it would make sense to avoid risks while scaling to both ways and rather scale too much upwards and less downwards.

Table 5.10 summarizes the steps taken in this cycle.

Cycle Step	**Action**
Diagnosing	Auto-scaling is not fast enough with scale up increment of one instance.
Planning and taking action	Test auto-scaling with scale up increment of two and four instances.
Evaluating	Server capacity was tripled within required five minutes with scale up increment of four instances.

Table 5.10: The action research cycle steps for Cycle 10

5.11 Summary of the Results

Table 5.11 summarizes the performance improvement results from Cycles 1-9. It does not include results from Cycle 10 as the results from auto-scaling cannot be compared to the results from performance increments made for a single instance.

Cycle	Requests per second	Performance improvement	Actions taken
1	8		Initial performance
2	10	25%	Double the computing power
3	19	90%	Use Node.js clusters
4	30	58%	Reduce amount of content rendered on the server
5	35	17%	Upgrade to React 16
6	70	100%	Upgrade to Node 8.8.1 and use more powerful instance type
7	80	14%	Use equal amount of Node.js clusters compared to the amount of CPU cores
8	90	13%	Upgrade instance type
9	800	789%	Skip rendering on the server if load is high

Table 5.11: Summary of actions taken and their performance improvements for Cycles 1-9

As Table 5.11 shows by far the most significant performance improvement came from Cycle 9 where we implemented the technique to skip rendering on the server if the load is high. The second most significant improvement came from Cycle 6. However, during Cycle 6 we took two actions that improved performance as we upgraded the Node.js version from 6.11.1 to 8.1.1 and upgraded the instance type from t2.medium to c4.xlarge.

According to Table 5.11, the third most significant improvement came from Cycle 3 where we implemented the use of Node.js cluster module. However, Node.js clusters themselves do not improve performance—they allow us to take full computing power out of each instance as by default Node.js runs only on a single CPU core. The last significant improvement is from Cycle 4 where the reduction of the content rendered on the server improved performance by 58%. For the rest of the cycles, performance improvements were less significant. However, it is worth to point out that Cycle 5 improved performance by 17% just by upgrading React version.

Chapter 6

Discussion

This chapter contains a discussion of the results of the research. The chapter is dived into three sections for each research question. The results are reflected against our own experience and the results found from literature.

6.1 RQ1: How to Ensure Availability of the Service Under High Load?

The primary objective of this thesis was to ensure the availability of our website under high load. The empiric research of this thesis focused on RQ2 and RQ3, which are part of the bigger picture pursued by RQ1.

Of the four techniques improving availability presented in Chapter 2, improvements from auto-scaling is discussed further in Section 6.3 and improvements from React application optimization in Section 6.2. Of the other two techniques, availability improvements of load balancing are mostly invisible. We are always running our server on at least two instances on two different availability zones in case another availability zone becomes unavailable. The role of load balancing as a technique to ensure availability cannot be dismissed as it is the technique which also allows auto-scaling to add more instances to the server infrastructure.

Content delivery network—Amazon CloudFront in our case—plays a critical role in our service availability. Our CloudFront logs show that at the start of 2019 only 7% of requests were not delivered from CloudFront cache. The logs also include application scripts served from CloudFront which increases CloudFront hit percent as the scripts are nearly always delivered from the cache. However, the logs show that the cache hit percent for most popular articles and the front page is over 80%. Therefore, CloudFront reduces the number of request to our servers considerably.

Availability of our website has been great mostly due to CloudFront since the end of May 2018 when the new site was also published for desktop users. The Iltalehti website has been unavailable only once for an hour and just for the part of our users. Unavailability was due to our actions, as we accidentally changed our CloudFront to point the wrong origin server. Therefore, no requests went from CloudFront to our server for an hour. However, only 5% of requests failed as CloudFront was able to serve old content from its cache. Therefore, our website worked for most of the users although users received old content on their first page loads which certainly was not desired effect. Still, CloudFront cache saved us from an even bigger accident.

Another incident since the launch of our responsive website was a distributed denial-of-service (DDoS) attack at the start of August 2018. During the attack, CloudFront received roughly 20 times more requests than usual. The attack lasted about 40 minutes, but the availability of our website was not affected. Response times on the server increased but not dramatically as for the top 10 percent of requests response times were over 2 seconds.

The DDoS attack happened early in the morning, and no one reported any issues during it. The whole incident was accidentally noticed one week after it happened. Since then we have improved our alarms to get notified right away once high traffic starts. During the attack, most of the traffic came from Asia and the United States. If we had received alarms of the high traffic, we could have blocked the traffic from those countries by using the geographic restrictions of CloudFront. Geo-blocking would mean that real users from affected countries could not access our website. We would still rather block those users if it meant that our service availability would be great in the rest of the countries and especially in Finland. According to our Google Analytics data, almost 96 percent of our traffic comes from Finland. Therefore, ensuring that our website works in Finland is the most crucial objective.

In our opinion, we have reached our main objective well. So far within the ten months, our service availability for all users has been 99.98% considering one-hour downtime for some users. Our service availability was not affected by a DDoS attack, and we have prepared for upcoming attacks with better notifications and Geo-blocking setup. We have also improved our server performance as discussed in the next section and our server auto-scaling as explained in the section after next.

6.2 RQ2: How to Improve Server-Side Rendering Performance of a React Application?

Our second objective was that our server could serve at least 1000 requests per second at any time. We knew we could reach that by simply multiplying our server capacity. However, we did not want just to pay more for increased capacity but also find other, more cost-efficient, solutions to our performance issues.

During our research, we managed to increase the performance of a single instance from the initial eight requests per second to 800 requests per second. Some of the performance increase was achieved by upgrading the instance type from t2.micro to c5.xlarge. The price of running a single c5.xlarge instance is about 15 higher compared to the t2.micro instance but still, the cost (276 euros per month) of running two c5.xlarge instances was acceptable for us. We could have reached our objective by running three c5.large instances instead of two c5.xlarge instances and save 70 euros per month, but we instead decided to pay some extra and have some additional buffer in our server performance.

The most significant technical solutions that increased our server performance were client-side rendering fallback, upgrades for Node.js and React versions and reductions of DOM content rendered on the server. The DOM content reduction was our idea invented based on the concept of component caching libraries. It provided even surprisingly good results as reducing content rendered on the server by nearly 50% increased server performance by over 50%. The only small drawback of the technique is that now the client needs to first re-render server content, and then render again to show the whole content.

Upgrading Node.js and React versions were recommended by many blog posts [11] [1] [7]. The upgrade of the React version did not yield us as good results as reported by others. Our server performance was only increased by 17% whereas for example, Arkwright [7] reported 25% improvement. Still, those results are closer to each other compared to three times faster server rendering reported by Aickin [1]. Then, server-side rendering and server performance do not mean the same thing, as the server also does other calculations in addition to rendering, although in our case rendering takes the biggest CPU time.

Due to memory issues and the upgrade of instance type, we do not have the exact percentage of how much Node.js version upgrade increased our server performance. However, we can estimate that upgrading from Node.js 6.11.1 to 8.8.1 almost doubled our server performance as with Node.js 6.11.1 the server could not serve 40 requests per second on the fifth stress test, and with Node.js 8.8.1 the server was able to serve closer over 70 requests per second with acceptable response times. This was even bigger performance improvement than what Arkwright [7] and Aickin [1] reported. At least a part of the bigger performance increase comes from our use of Node.js 8.8.1. Arkwright and Aickin both did their tests using Node.js 8.4.0, and we saw a big performance increase between versions 8.4.0 and 8.8.1 in our tests.

Client-side rendering fallback—the idea presented by Arkwright [7]—gave us by far the most significant performance increment. It increased the number of requests our server was able to serve by slightly over eight times. The result was very similar to Arkwright's, as Arkwright reported eight times higher performance in his benchmarks. The client-side rendering fallback comes with a price as skipping server-side rendering slows the first page load and disables some search engines from crawling our website. However, the client-side rendering fallback is only meant to be a temporary solution to keep the site running under high load, and we can pay the price of missing SEO benefits and slowing first page load if it takes that to keep the website functional.

Out of the React optimization techniques presented in Section 2.5, we are utilizing all but component caching. Even though some [17] [7] reported significant performance improvements from the component caching and our website could have significantly benefited from it—as our site has same content repeated on all pages (for instance header, navigation, and footer)—we decided not to pursue component caching. We considered that third-party libraries changing the React private APIs are too high risk for future development as changes made by the libraries may not be compatible with newer versions of React. That would have left us to the situation where we waited for the third-party libraries to be updated or where we made changes to them ourselves or where we did not upgrade React at all. All these options would slow our development, and therefore, we did not even try to use component caching.

6.3 RQ3: How to Auto-Scale Quickly Once High Load Starts?

Our third objective was that our server must be able to scale to serve three times higher amount of request than it usually can within 5 minutes from the start of high load. Out of the considered auto-scaling trigger options, only CPU utilization turned out to be suitable. Latency was not an appropriate trigger option as due to the client-side rendering fallback average latency does not correlate with the load on the server. We were also unable to get Beanstalk to scale based on request count. On the other hand, CPU utilization is a more flexible trigger option than request count as with request count we should have estimated how many requests our server can handle before we need to add more capacity. Then we should update that number as the performance of the server changes.

We did not find any reason to change the chosen statistic (average CPU utilization) or the upper and lower thresholds (75% and 35% respectively) as they worked well through the auto-scaling tests. The period between metric evaluations was set to one minute which is the lowest possible value in Beanstalk. As the breach duration was set to two minutes, scaling up might take almost three minutes if the first metric evaluation was made just before the start of the increased traffic. We were slightly disappointed in scaling being that slow, but as the time between evaluations could not be shortened, the only option would have been to decrease the breach duration to one minute. As we were evaluating average CPU utilization, scaling after one minute could have been reasonable, but we still feared that it could cause too many unnecessary scale-ups.

In the end, we only ended up changing the number of instances added or removed during each scaling event. As the trigger action could take up to three minutes and adding instances about 40 seconds, the only way to triple server capacity within five minutes was to directly triple the number of instances on the first scale up event. In our case, we had to add four new instances in addition to the two already running. Adding four new instances could risk over-provisioning, but as the costs of running instances for a short time are low, as presented earlier in Table 3.2, it makes sense for us to pay some extra to improve performance, and thus availability. For example, running ten c5.xlarge instances for an hour costs only about 2 euros. Therefore, running even 100 instances for a short time would be reasonable if it takes that to keep the site running during an extremely high load.

Chapter 7

Conclusions

Availability of the Iltalehti website is a significant concern for Iltalehti. The primary objective of this thesis was to ensure availability of Iltalehti website under the high load. Reflecting on the results and the discussion, it can be concluded that the objective was reached well. The four main techniques to ensure and improve availability for case Iltalehti were load balancing, content delivery networks, auto-scaling and enhancing the performance of server-side rendered React application. All were researched with literature review and auto-scaling and server performance also through empirical research.

The availability improvement provided by load balancing is the least visible of the four. Balancing the load to different availability zones protects from instance hardware failures and power outages. If an error occurs on a certain availability zone, load balancer directs traffic to instances running on the other zones.

For a website serving mostly static content, content delivery network (CDN) is an essential tool improving availability. As most requests are made to specific resources—such as the front page—those resources can almost always be returned from the cache of the CDN. CDN also provides some protection against DDoS attacks as requests to the same resource do not cause a request to the server. Availability is also improved by CDN being able to serve old, cached content in case the server does not respond.

Improvement of server performance improves service availability as the server can serve more request before becoming overwhelmed. The server performance of a single instance was increased from 8 requests per second to 800 requests per second during the study. The improvement was a result of both upgrading to the more powerful instance and improvements in the React application. The most significant improvement techniques related to React were the client-side rendering fallback, upgrades for Node.js and React versions and reductions of content rendered on the server. Especially the client-side rendering fallback—a technique where server-side rendering was skipped, if the request rate was over the selected limit—provided significant performance improvement as it enabled the server to serve eight times more requests per second.

Auto-scaling allows cost-effectively to run only server capacity that is needed. The auto-scaling process to add more instances has to happen quickly once the load starts increasing. During this research, an auto-scaling process in Elastic Beanstalk took 3-4 minutes. To reach the objective of trebling server capacity within five minutes, the auto-scaler had to add three times more instances in a single scaling process.

One interesting topic for further research would be the use of serverless architecture instead of the traditional server architecture. The serverless architecture could improve availability as the computing power of the server would not be a bottleneck. Each request would be handled on its own, and there would be no need to configure auto-scaling as serverless can handle any number of requests. However, the costs of running serverless, logging and monitoring would need to be studied further before changing to technique.

Bibliography

[1] Aickin, S. What's new with server-side rendering in react 16, 2017. WWW blog post of Sasha Aickin: https://hackernoon.com/whats-new-with-server-side-rendering-in-react-16-9b0d78585d67. Accessed 17 Sep 2018.

[2] Alakeel, A. M. A guide to dynamic load balancing in distributed computer systems. *International Journal of Computer Science and Information Security 10*, 6 (2010), 153–160.

[3] Amazon Web Services. Monitoring memory and disk metrics for amazon ec2 linux instances, 2017. WWW page of the AWS Elastic Beanstalk: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/mon-scripts.html. Accessed 12 Dec 2017.

[4] Amazon Web Services. Amazon ec2 c5 instances, 2018. WWW page of Amazon Web Service: https://aws.amazon.com/ec2/instance-types/c5/. Accessed 5 Aug 2018.

[5] Amazon Web Services. Amazon ec2 pricing, 2018. WWW page of Amazon Web Service: https://aws.amazon.com/ec2/pricing/on-demand/. Accessed 15 Dec 2018.

[6] Amazon Web Services. Aws elastic beanstalk features, 2018. WWW page of Amazon Web Service: https://aws.amazon.com/elasticbeanstalk/details/. Accessed 5 Aug 2018.

[7] Arkwright. Scaling react server-side rendering, 2017. WWW page of Arkwright: https://arkwright.github.io/scaling-react-server-side-rendering.html. Accessed 29 Feb 2018.

[8] Bauer, E., and Adams, R. *Reliability and availability of cloud computing.* John Wiley & Sons, 2012.

[9] Chaczko, Z., Mahadevan, V., Aslanzadeh, S., and Mcdermid, C. Availability and load balancing in cloud computing. In *International Conference on Computer and Software Modeling, Singapore* (2011), vol. 14.

[10] Chieu, T. C., Mohindra, A., Karve, A. A., and Segal, A. Dynamic scaling of web applications in a virtualized cloud computing environment. In *E-Business Engineering, 2009. ICEBE'09. IEEE International Conference on* (2009), IEEE, pp. 281–286.

[11] Clark, A. React v16.0, 2017. WWW blog post of Andrew Clark: `https://reactjs.org/blog/2017/09/26/react-v16.0.html`. Accessed 17 Sep 2018.

[12] Coghlan, D., and Brannick, T. *Doing action research in your own organization.* Sage, 2014.

[13] Di Lucca, G. A., and Fasolino, A. R. Testing web-based applications: The state of the art and future trends. *Information and Software Technology 48*, 12 (2006), 1172–1186.

[14] Dunn, J., and Crosby, B. What your cdn won't tell you: Optimizing a news website for speed and stability. In *LISA* (2012), pp. 195–201.

[15] Facebook Inc. React, 2018. WWW www page of Facebook Inc.: `https://reactjs.org/`. Accessed 17 Oct 2018.

[16] Ghanbari, H., Simmons, B., Litoiu, M., and Iszlai, G. Exploring alternative approaches to implement an elasticity policy. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on* (2011), IEEE, pp. 716–723.

[17] Grigoryan, A. Using electrode to improve react server side render performance by up to 70 WWW blog post of Alex Grigoryan: `https://medium.com/walmartlabs/using-electrode-to-improve-react-server-side-render-performance-by-up-to-70-e43f9494eb8b`. Accessed 17 Sep 2018.

[18] Jamae, J. Response time vs. latency, 2005. WWW blog post of Javid Jamae: `http://www.javidjamae.com/2005/04/07/response-time-vs-latency/`. Accessed 10 Oct 2018.

[19] Krishnamurthy, D., Rolia, J. A., and Majumdar, S. A synthetic workload generation technique for stress testing session-based systems. *IEEE Transactions on Software Engineering 32*, 11 (2006), 868–882.

[20] Lorido-Botran, T., Miguel-Alonso, J., and Lozano, J. A. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing 12*, 4 (2014), 559–592.

[21] Menascé, D. A. Load testing of web sites. *IEEE Internet Computing 6*, 4 (2002), 70–74.

[22] Nelapati, Y. Scaling up with nginx, 2011. WWW page of Yash Nelapati: `https://yashh.com/scaling-up-with-nginx`. Accessed 29 Mar 2018.

[23] Node.JS. Cluster — node.js v6.15.1 documentation, 2018. WWW documentation of Node.JS: `https://nodejs.org/docs/latest-v6.x/api/cluster.html`. Accessed 28 Jan 2018.

[24] Pallis, G., and Vakali, A. Insight and perspectives for content delivery networks. *Communications of the ACM 49*, 1 (2006), 101–106.

[25] Rangel, F. 4 practical tips for drastically improved server-side rendering in react, 2016. WWW blog post of Freddy Rangel: `https://medium.com/react-university/4-practical-tips-for-drastically-improved-server-side-rendering-in-react-2df98555a26b`. Accessed 17 Sep 2018.

[26] ReactCC. Speedier server-side rendering in react 16 with component caching, 2018. WWW blog post of ReactCC: `https://medium.com/@reactcomponentcaching/speedier-server-side-rendering-in-react-16-with-component-caching-e8aa677929b1`. Accessed 17 Sep 2018.

[27] Susman, G. I., and Evered, R. D. An assessment of the scientific merits of action research. *Administrative science quarterly* (1978), 582–603.

[28] Tarvainen, J. The performance cost of server side rendered react on node.js, 2017. WWW blog post of Jani Tarvainen: `https://malloc.fi/performance-cost-of-server-side-rendered-react-node-js`. Accessed 17 Sep 2018.

[29] TNS Metrix. Suomen web-sivustojen viikkoluvut, 2017. WWW www page of TNS Metrix: `http://tnsmetrix.tns-gallup.fi/public/`. Accessed 27 Feb 2018.

[30] Triukose, S., Al-Qudah, Z., and Rabinovich, M. Content delivery networks: protection or threat? In *European Symposium on Research in Computer Security* (2009), Springer, pp. 371–389.

Appendix A

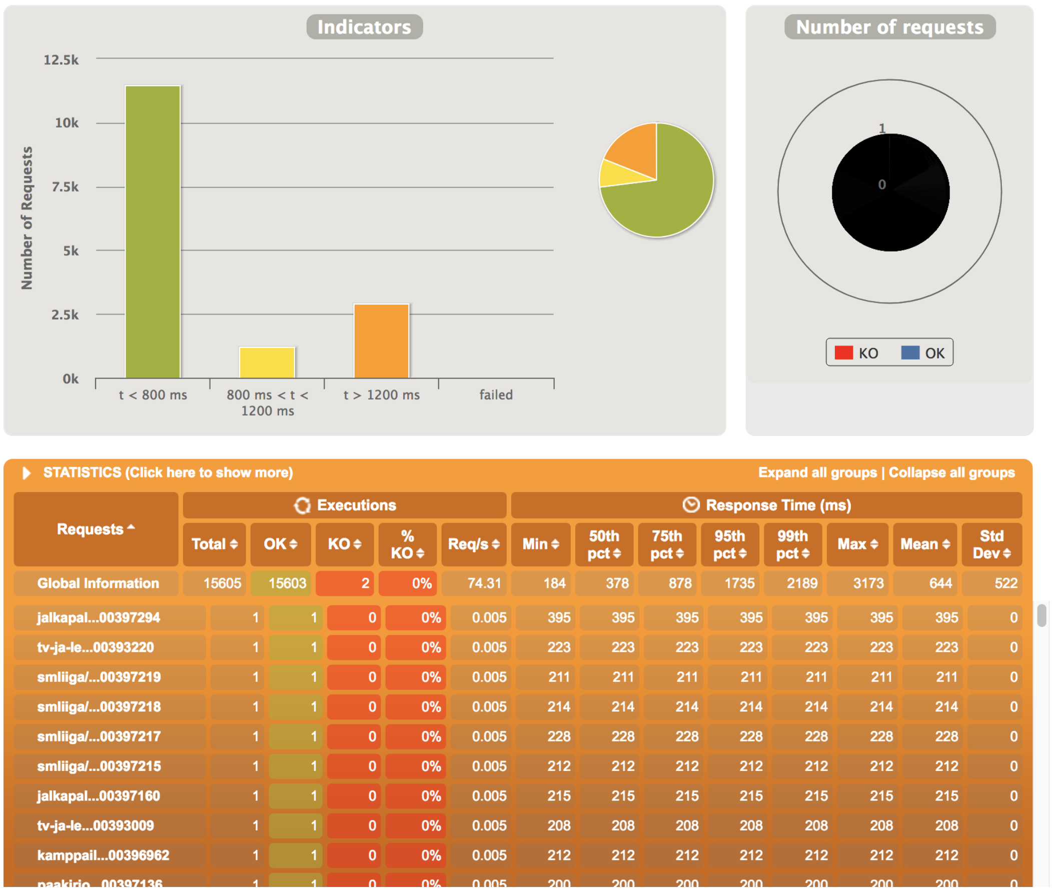
Gatling Stress Test Simulation Code

```scala
import scala.concurrent.duration._
import io.gatling.core.Predef._
import io.gatling.http.Predef._
import io.gatling.jdbc.Predef._

class Falconfttest extends Simulation {

    val httpProtocol = http
        .baseURL("https://stresstest.iltalehti.fi/")
        .acceptEncodingHeader("gzip, deflate, sdch")
        .acceptLanguageHeader("fi-FI,fi;q=0.8,en-US;q=0.6,en;
            q=0.4")
        .userAgentHeader("Mozilla/5.0 (Macintosh; Intel Mac
            OS X 10_12_6) AppleWebKit/537.36 (KHTML, like
            Gecko) Chrome/57.0.2987.98 Safari/537.36")

    val headers_1 = Map(
        "Accept" -> "text/html,application/xhtml+xml,
            application/xml;q=0.9,image/webp,*/*;q=0.8",
        "Accept-Encoding" -> "gzip, deflate, sdch, br",
        "Connection" -> "keep-alive",
        "Upgrade-Insecure-Requests" -> "1")

    val rps = Integer.getInteger("rps", 20)
    val reachInSec = Integer.getInteger("reachInSec", 60)
    val holdForMin = Integer.getInteger("holdForMin", 2)
    val urlFeed = System.getProperty("urlFeed", "urls.csv")
    println("================================================
        ")
    println(s"Running Falcon test with feed: $urlFeed")
    println(s"$rps requests per seconds")
    println(s"Reach in $reachInSec seconds")
    println(s"Hold for $holdForMin minutes")
    println("================================================
       ")
    /*Usage example:
    export JAVA_OPTS="-DurlFeed=broken_article_urls.csv -Drps
       =30 -DreachInSec=30 -DholdForMin=1"
    sudo bin/gatling.sh
    */

    val feeder = csv(urlFeed).circular

    val scn = scenario("FalconSimulation")
        .feed(feeder)
        .exec(http("${urls}")
            .get("${urls}")
            .headers(headers_1))

    setUp(
        scn.inject(
            constantUsersPerSec(1000) during(10 minutes)
        )
    ).throttle(
        reachRps(rps) in (reachInSec seconds),
        holdFor(holdForMin minutes)
    ).protocols(httpProtocol)
}
```

Listing A.1: Gatling stress test simulation code

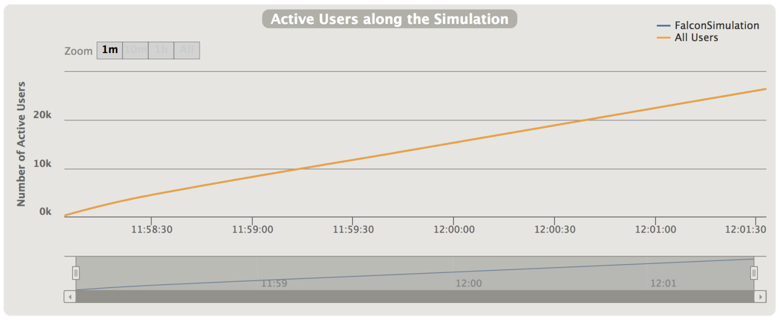
Appendix B

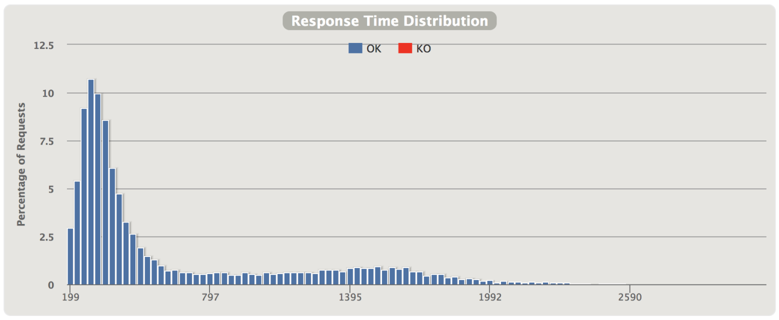
An Example of a Gatling Stress Test Report

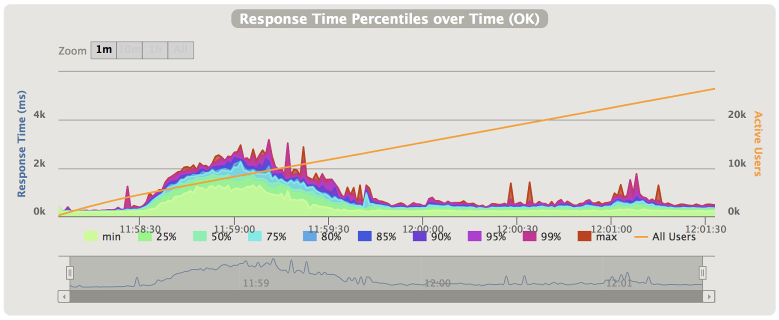


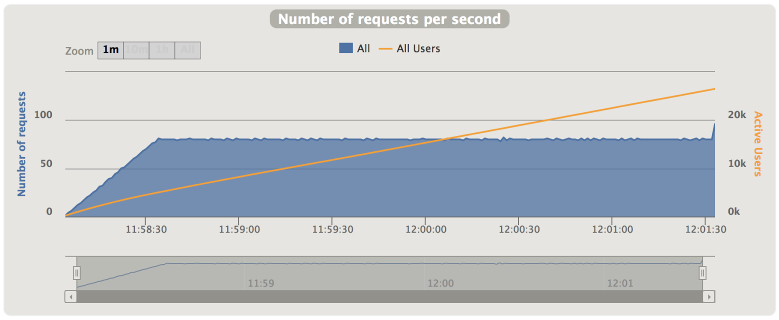
STATISTICS (Click here to show more) Expand all groups | Collapse all groups

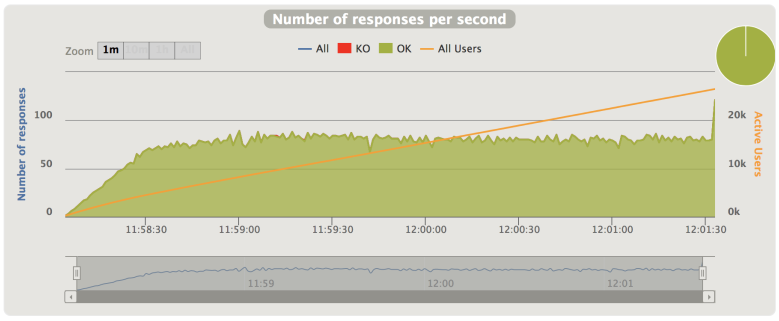
Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	15605	15603	2	0%	74.31	184	378	878	1735	2189	3173	644	522
jalkapal...00397294	1	1	0	0%	0.005	395	395	395	395	395	395	395	0
tv-ja-le...00393220	1	1	0	0%	0.005	223	223	223	223	223	223	223	0
smliiga/...00397219	1	1	0	0%	0.005	211	211	211	211	211	211	211	0
smliiga/...00397218	1	1	0	0%	0.005	214	214	214	214	214	214	214	0
smliiga/...00397217	1	1	0	0%	0.005	228	228	228	228	228	228	228	0
smliiga/...00397215	1	1	0	0%	0.005	212	212	212	212	212	212	212	0
jalkapal...00397160	1	1	0	0%	0.005	215	215	215	215	215	215	215	0
tv-ja-le...00393009	1	1	0	0%	0.005	208	208	208	208	208	208	208	0
kamppail...00396962	1	1	0	0%	0.005	212	212	212	212	212	212	212	0
paakirjo...00397136	1	1	0	0%	0.005	200	200	200	200	200	200	200	0


Active Users along the Simulation
FalconSimulation
All Users
Zoom 1m 10m 1h All
Number of Active Users
20k
10k
0k
11:58:30
11:59:00
11:59:30
12:00:00
12:00:30
12:01:00
12:01:30
11:59
12:00
12:01


Response Time Distribution
OK
KO
Percentage of Requests
12.5
10
7.5
5
2.5
0
199
797
1395
1992
2590


Response Time Percentiles over Time (OK)
Zoom 1m 10m 1h All
Response Time (ms)
4k
2k
0k
Active Users
20k
10k
0k
11:58:30
11:59:00
11:59:30
12:00:00
12:00:30
12:01:00
12:01:30
min
25%
50%
75%
80%
85%
90%
95%
99%
max
All Users
11:59
12:00
12:01


Number of requests per second
Zoom 1m 10m 1h All
All
All Users
Number of requests
100
50
0
Active Users
20k
10k
0k
11:58:30
11:59:00
11:59:30
12:00:00
12:00:30
12:01:00
12:01:30
11:59
12:00
12:01


Number of responses per second
Zoom 1m 10m 1h All
All
KO
OK
All Users
Number of responses
100
50
0
Active Users
20k
10k
0k
11:58:30
11:59:00
11:59:30
12:00:00
12:00:30
12:01:00
12:01:30
11:59
12:00
12:01