Cloud Computing

Horizontal Scaling

Stefan Lont (s1186053),
Martijn Schuman (s1186586),
Jasper van Willigen (s1186837),
Jorrick Stempher (s1172166)

Abstract:

This study focuses on identifying the most suitable node specifications at Vultr to optimize the performance and cost of a microservice application. The research question centers on determining which node pool type scales optimally during horizontal scaling within Kubernetes. To answer this, three sub-questions were formulated: identifying available node pool types and plans, determining the most suitable node type per application layer, and assessing performance under different load levels.

Through literature review, prototyping, and load testing with K6, both Regular Performance and High Performance (AMD EPYC and Intel Xeon) nodes were tested. Additionally, the effect of the vCPU-to-instance ratio on resource utilization and maximum load was analyzed. The tests showed that:

1. High Performance (Intel Xeon) nodes manage the highest number of requests per second (RPS: 1900-7200) and are thus most suitable for both the client and API layers.
2. Splitting a vCPU across multiple instances results in inefficient resource usage and lower maximum load, while one instance per vCPU ensures more stable performance and better utilization of CPU and RAM resources.
3. Limited multithreading in Next.js and inefficient handling of short CPU peaks contribute to performance limitations in configurations with multiple instances per vCPU.

The findings provide a foundation for cost-conscious scaling strategies and emphasize the importance of balancing vCPUs and instances for optimal performance. Future research in a controlled environment is recommended to further validate the reliability of the test results.

Introduction

This research aims to determine the optimal node specifications for Vultr to improve the efficiency of a developed application. A node, in this case a virtual machine (VM), forms the foundation of the infrastructure.

The research follows several phases. In the first phase, Vultr's available node pool plans are mapped out. A node pool type is a category that includes various node pool plans, each with unique combinations of CPU, RAM, storage capacity, bandwidth, and cost.

Next, the most suitable node type for different layers of the application, such as the Next.js client and Nest.js API, is analyzed. The final phase investigates the number of requests each node specification can manage.

The results of this research contribute to reducing costs for virtual machines without compromising application performance. Additionally, the findings serve as a foundation for follow-up research aimed at predicting cluster load and automating horizontal scaling within a Kubernetes cluster.

The research was conducted for the assignment, which is described in Appendix 1: The assignment.

Research Questions

The research was guided by a main question, supported by three sub-questions:

Main question

Which node pool type at Vultr is desirable for scaling a node pool?

Sub-questions

- What node pool types are available at Vultr.
- Which node type is most advantageous for different application layers?
- How many requests can the node plan of the selected node type handle for the different application layers?

Methods

Literature Review

This research utilizes a literature review to uncover Vultr's configuration options for Kubernetes nodes. Vultr's documentation was consulted to answer the first sub-question.

The collected information forms the foundation for developing prototypes and grouping data for analysis.

Various options are summarized in a table to provide a clear overview.

Prototype Development

The second method involved building a prototype to collect data for analysis. This prototype includes a Kubernetes cluster hosted at Vultr, containing various worker pools, each with a fixed node type and a specific node pool plan. Only nodes of the specified type can be added to the pool.

Cluster Setup

The prototype includes a Kubernetes cluster set up at the hosting provider Vultr. Within this cluster, various worker pools are managed. Each worker pool has a fixed node type and a specific node pool plan. Only nodes of that type can be added to the pool.

Figure 1 Node pools within Kubernetes Cluster at Vultr shows two node pools with different node specs.
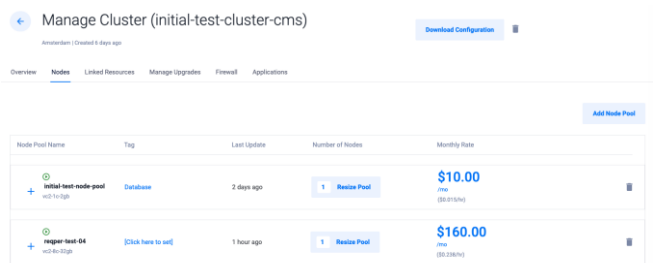

Figure 1 Node pools within Kubernetes Cluster at Vultr

Cluster Setup

Before the tests could be performed, the client and API were installed separately on the cluster. External actors, such as Cloudflare and the database, were considered to prevent unnecessary load on these systems.

For the client, the API is simulated by consistently using the same dataset. After analyzing several SQL queries on the database, it was decided to introduce a built-in delay of 30 milliseconds.

The 30-millisecond delay is based on average query times measured during previous tests. This ensures realistic load and prevents the simulation from producing unrealistically fast or inconsistent results. The full justification for this choice is included in Appendix 2: API-query times.

For the API, the database and Cloudflare were simulated. Unlike the front-end, the database has no added delay. This is because the query times were so low that an additional delay would not noticeably affect the measured values, as explained in Appendix 3: Database query timing. Cloudflare was simulated by

returning fake image URLs, meaning no connection was made to Cloudflare.

Generating server load

To obtain test data, load tests are conducted on various node plans. These tests are conducted using the K6 simulator, a tool developed by Grafana for performing load tests on applications, including Kubernetes nodes (Grafana, 2024).

Using K6 it's possible to:
- Specify the number of Virtual Users (VUs).
- Configure HTTP endpoints and test intervals.
- Set thresholds.

In consultation with the client, the following thresholds were established for valid tests:
- 0% of HTTP requests may be lost.
- 95% of the HTTP requests must be answered within 1000ms.

To ensure a reliable load test, K6 was used with changing amount of VUs and a ramp-up period of five seconds, gradually increasing server load before running a one-minute instantLoad test. Detailed configuration and results are provided in Appendix 4: K6 example.

Collecting Information

After conducting the tests, the Requests Per Second (RPS) can be calculated using Equation 1. RPS provides a simple way to compare different nodes with each other. Here:

- VU is the number of virtual users used,
- T is a fixed value of one second, and
- E is the number of tested endpoints (two for the API and three for the client).

$$RPS = \frac{VU \times E}{T}$$

Equation 1 Formula for calculating Requests Per Second

By adjusting the variable VU during the tests, the limit of a node can be determined. When a test passes, the number of VUs is increased by 100. If a test fails, it is repeated to verify whether the failure might have been caused by an edge case.

An edge case refers to a scenario where a test fails, but the specific cause of the failure is not immediately clear. Various factors could contribute to such unclear errors during node testing, including unexpected network delays, traffic spikes, node resource limits, the performance of the device running K6, or even temporary failures in the underlying infrastructure. Since some of these factors are beyond control,

pinpointing the exact cause of a failure can be challenging.

If a test fails again upon repetition, the number of VUs is decreased by 50 in subsequent tests until the test passes. This process helps identify the node's limits and determines the number of VUs at which the test is successfully executed. To ensure that a node can manage a certain load consistently, the test is repeated four times with the reduced VU count. If all four repetitions pass, the breakpoint is confirmed.

Typically, a test fails when the CPU becomes overloaded and can no longer process the high volume of incoming HTTP requests within the 1000ms threshold.

Detailed Test Results

All test results are meticulously recorded in an Excel file to facilitate later comparisons and analysis. For each test, the following data points are stored:
- **TestID:** A unique identifier for the test.
- **Node Name:** The name of the node tested, formatted as follows: [node type abbreviation]-test[number]-[application].
- **VM-Spec:** A reference to the tested node's specifications.
- **Instances:** The number of application instances used for the test (e.g., one instance per vCPU).
- **Test Type:** The type of test script used.
- **Set Virtual Users:** The number of virtual users configured for the test.
- **Endpoints:** The HTTP endpoints targeted during the test.
- **Threshold:** The threshold value the test must meet.
- **Ramp-Up Time:** The time taken to gradually increase the number of virtual users.
- **Max Test Time:** The maximum duration allowed for the test.
- **Virtual Users Created**: The actual number of virtual users created during the test.
- **Total Test Time:** The total duration of the test.
- **Total Requests Sent:** The total number of requests sent during the test.
- **Median:** The median response time for requests.
- **p(95):** The 95th percentile response time, indicating that 95% of requests were faster than this value.
- **Lost Requests:** The number of requests that failed to send or receive successfully.
- **Died:** Whether the test passed without exceeding the threshold.

- **Total Requests Received:** The number of requests successfully received and processed by the server.

These data points provide detailed insights into the performance of both the client and the API, as well as the server's ability to handle different levels of load. They also enable a comprehensive comparison of various test scenarios, aiding in identifying bottlenecks and optimizing performance.

Identifying the Best Node Type

At the conclusion of these tests, it will be clear which node type best suits each application layer. However, this data alone is insufficient to fully answer the research question.

For this reason, the first three node plans within each chosen node type will also be tested. By determining the capabilities of these node plans, the results can be extrapolated to predict the performance of all node plans for the selected node types.

Data Analysis

To analyze the collected test data, all test results were documented and organized in an Excel file. This file contains all parameters as described in Detailed Test Results. Structuring the data enabled various statistical analyses and comparisons to be performed.

Figure 2 Test Results API (High Performance, Intel)

Step 1: Structuring the Data

The raw test data was entered into Excel using clear column names for all recorded parameters (e.g., test number, node name, number of virtual users, and thresholds). A consistent format was applied to minimize errors during analysis.

Step 2: Summarizing the Results

To effectively compare the performance of various node types and node plans, only the most relevant information from the raw test data was retained. These key metrics were structured into a concise summary, forming the foundation for visualizations in Step 3. An example of this summarized data is presented in Figure 3.

Figure 3 Summarized Test Data (High Performance, Intel)

Step 3: Visualising the Results

To better interpret the summarized data, several types of charts were created in Excel:

- **Bar Charts**: Provide a clear overview of the average RPS (Requests Per Second) for each node type across different application layers under varying numbers of virtual users.
- **Line Graphs**: Highlight trends in average response times for each node plan, enabling an easy comparison of performance over time or with increasing loads.

These visualizations make it easier to identify patterns, bottlenecks, and areas for optimization across the tested configurations. They also serve as a key resource for decision-making regarding node selection and infrastructure planning.

Results
What Node Pool Types and Plans Are Available at Vultr?

Before determining the most efficient pairing of nodes and application layers, all node options offered by hosting provider Vultr were reviewed.

Vultr provides eight different node pool types for Cloud Computing. These node pools include nodes with progressively increasing specifications in:

- **Virtual processors (vCPUs)**: A portion of an underlying physical processor (Datacenter.com, 2024).
- **Memory (RAM)**: The amount of volatile memory available.
- **Bandwidth**: The network traffic capacity per month.
- **Storage**: Disk space for data.
- **Cost**: Pricing based on the above specifications.

Additionally, Vultr offers a separate category of virtual machines with enhanced CPU or memory performance at higher prices. These machines were excluded from this study, as the expected number of requests to the developed application is too low to justify the cost Appendix 5: Expected .

The following types of nodes were also omitted from the study for reasons of practicality and relevance:

Storage-optimized nodes: These are tailored for data-heavy applications, which is not the focus of the application being evaluated.

Nodes with less than 1GB of RAM: According to (Hayhurst, 2022) these nodes lack the capacity to run the developed application effectively.

By narrowing down the options to only relevant node pool types, the study focused on configurations that would realistically meet the application's performance and cost requirements. This approach ensures that the findings can guide optimal infrastructure decisions while avoiding unnecessary complexity.

Cloud Compute types

Regular Performance

The Regular Performance node type utilizes an unspecified Intel CPU paired with a standard SSD. The various node plans for this type are listed in Table 1.

These nodes are designed to offer a balanced combination of processing power and storage for standard cloud computing tasks. Due to the unspecified CPU, the specific capabilities may vary, but they generally provide a reliable foundation for applications that do not require high-end performance or specialized hardware. The use of a standard SSD ensures sufficient storage speed for general purposes, though it may not perform as well for data-intensive applications.

vCPU's	Memory	Bandwidth	Storage	Price
1	1 GB	1 TB	25 GB	$0,007/h
1	2 GB	2 TB	55 GB	$0,015/h
2	2 GB	3 TB	65 GB	$0,022/h
2	4 GB	3 TB	80 GB	$0,03/h
4	8 GB	4 TB	160 GB	$0,06/h
6	16 GB	5 TB	320 GB	$0,12/h
8	32 GB	6 TB	640 GB	$0,24/h
16	64 GB	10 TB	1280 GB	$0,48/h
24	96 GB	15 TB	1600 GB	$0,95/h

Table 1: Node Plans for Regular Performance Type

High Performance

The High-Performance node type offers a choice between two different CPUs:
- AMD EPYC (**Fout! Verwijzingsbron niet gevonden.**)
- Intel Xeon (**Fout! Verwijzingsbron niet gevonden.**)

Other than the CPU, the specifications of the nodes are identical, including the price.

AMD EPYC

vCPU's	Memory	Bandwidth	Storage	Price
1	2 GB	3 TB	50 GB	$0,018/h
2	2 GB	4 TB	60 GB	$0,027/h
2	4 GB	5 TB	100 GB	$0,036/h
4	8 GB	6 TB	180 GB	$0,071/h
4	12 GB	7 TB	260 GB	$0,107/h
8	16 GB	8 TB	350 GB	$0,143/h
12	24 GB	12 TB	500 GB	$0,214/h

Table 2: Overview of High-Performance Node Plans (AMD)

Intel Xeon

vCPU's	Memory	Bandwidth	Storage	Price
1	2 GB	3 TB	50 GB	$0,018/h
2	2 GB	4 TB	60 GB	$0,027/h
2	4 GB	5 TB	100 GB	$0,036/h
4	8 GB	6 TB	180 GB	$0,071/h
4	12 GB	7 TB	260 GB	$0,107/h
8	16 GB	8 TB	350 GB	$0,143/h
12	24 GB	12 TB	500 GB	$0,214/h

Table 3: Overview of High-Performance Node Plans (Intel)

Which Node Type is Most Suitable for the Different Application Layers?

To ensure cost-efficient scaling for the developed CMS application, tests were conducted to identify the most suitable node type for its two application layers: the client and the API. These tests were performed under consistent threshold values and conditions as described in the Prototype Development chapter.

Regular Performance

For the Regular Performance node type, the node plan detailed in Table 4 was tested:

vCPU's	Memory	Bandwidth	Storage	Price
1	2 GB	2 TB	55 GB	$0,015/h

Table 4 Specifications and costs of the Regular Performance node plan.

The breakpoints of this node plan were determined for both application layers:
- For the client, the breakpoint was reached at 150 simultaneous virtual users (VU), corresponding to a Request Per Second (RPS) of 450.
- For the API, the breakpoint was reached at 300 simultaneous VUs, corresponding to an RPS of 600.

App	Endpoints (E)	Breakpoint (VU)	RPS
Client	3	150	450
API	2	300	600

Table 5: Performance results for the Regular Performance node plan.

High Performance AMD EPYC

For the High Performance (AMD EPYC) node type, the node plan shown in Table 6 was tested:

vCPU's	Memory	Bandwidth	Storage	Price
2	2 GB	3 TB	50 GB	$0,027/h

Table 6: Specifications and costs of the High Performance (AMD EPYC) node plan.

The breakpoints of this node plan were as follows:
- For the client, the breakpoint was reached at 400 simultaneous VUs, corresponding to an RPS of 1200.
- For the API, the breakpoint was reached at 850 simultaneous VUs, corresponding to an RPS of 1700.

App	Endpoints (E)	Breakpoint (VU)	RPS
Client	3	400	1200
API	2	850	1700

Table 7: Performance results for the High Performance (AMD EPYC) node plan.

High Performance Intel Xeon

For the High Performance (Intel Xeon) node type, the node plan described in was tested:

vCPU's	Memory	Bandwidth	Storage	Price
2	2 GB	3 TB	50 GB	$0,027/h

Table 8: Specifications and costs of the High Performance (Intel Xeon) node plan.

The breakpoints for this node plan were determined as follows:
- For the client, the breakpoint was reached at 475 simultaneous VUs, corresponding to an RPS of 1425.
- For the API, the breakpoint was reached at 950 simultaneous VUs, corresponding to an RPS of 1900.

App	Endpoints (E)	Breakpoint (VU)	RPS
Client	3	475	1425
API	2	950	1900

Table 9: Performance results for the High Performance (Intel Xeon) node plan.

Sub-Conclusion

Based on the test results and costs of the tested node plans, the product owner decided to limit further tests to the following three node types:
- Regular Performance
- High Performance AMD EPYC
- High Performance Intel Xeon

Other node types, such as high-frequency and optimized cloud compute nodes, were excluded from the scope of this research, as they are not as relevant or cost-effective for the CMS application.

The results indicate that the High-Performance Intel Xeon is the most suitable node type for the CMS client. This node type achieved the highest RPS capacities at a relatively low cost, making it ideal for scalable and cost-efficient deployments. Both the client and the API

reached their highest RPS values (1425 and 1900, respectively) with this node type.

While the High-Performance AMD EPYC node type offers similar pricing to the Intel Xeon, it performed slightly worse in terms of maximum RPS, making it less attractive for high-load applications. The Regular Performance nodes, although budget-friendly, are less suitable for intensive workloads due to their limited scalability and lower RPS.

This finding will be carried forward into the third research question, where the capabilities of more expensive node plans within the High-Performance Intel Xeon node type will be further explored for the client layer.

How Many Requests Can the Node Plans of the Chosen Node Type Handle for the Different Application Layers?

To scale the application as efficiently as possible, research was conducted on the performance of various node plans. The focus was on the node plans of the High Performance (Intel) type, which emerged as the best option in previous tests. This research aimed to optimize scaling for both the client and API layers of the developed application.

All tests were conducted under the same threshold values and test conditions, as described in the Prototype Development chapter.

Table 10 summarizes the three different node plans (NP) tested for this sub-question. All node plans belong to the High Performance (Intel) node type.

#NP	vCPU's	Memory	Bandwidth	Storage	Price
1	2	2 GB	3 TB	60 GB	$0,027/h
2	4	8 GB	6 TB	180 GB	$0,07/h
3	8	16 GB	8 TB	350 GB	$0,14/h

Table 10: Specifications and costs of the tested node plans under the High Performance (Intel) node type.

The breakpoints of the three node plans were determined for both application layers: the client and the API. The first (cheapest) node plan reached its breakpoint at 475 VUs for the client and 950 VUs for the API, resulting in an RPS of 1900 for both application layers.

As expected, the second node plan performed better: the breakpoint for the client was reached at 800 VUs,

while the API reached 2200 VUs. This corresponded to an RPS of 2400 for the client and 4400 for the API.

The final node plan delivered the best performance, with a breakpoint of 2700 VUs for the client and 3600 VUs for the API. The corresponding RPS was 2700 for the client and 7200 for the API.
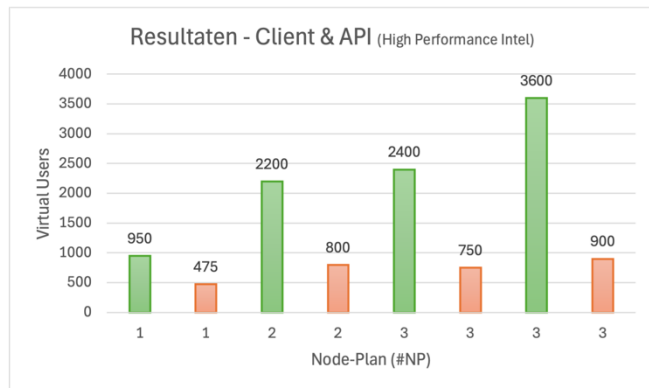

Figure 4: Bar chart for the VU Breakpoint (API green, client orange)

The test results for each node plan are shown in Figure 4 and Table 11, which both provide an overview of the performance for the two application layers per node plan.

#NP	App	Endpoints (E)	Breakpoint (VU)	RPS
1	Client	3	475	1900
1	API	2	950	1900
2	Client	3	800	2400
2	API	2	2200	4400
3	Client	3	900	2700
3	API	2	3600	7200

Table 11: Overview of the test results per application layer per node plan

In Figure 5 and Figure 6, the trends in the median and 95th percentile of the response times are visualized. These figures provide deeper insight into the average performance of each node plan.
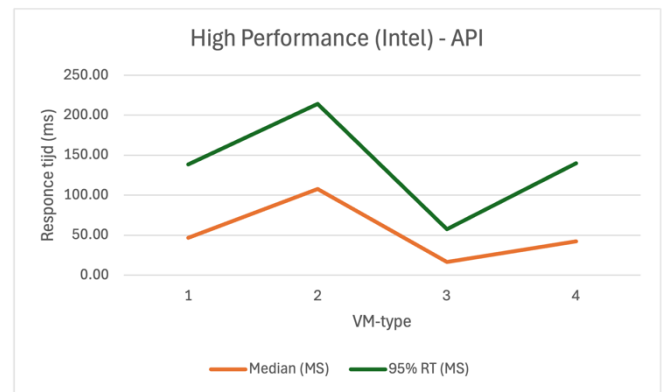

Figure 5: Medium & 95e percentile Client.

Figure 6: Medium & 95e percentile API.

Discussion
Additional Findings
Ramp-up
The initial load tests were conducted without a ramp-up time, meaning that the total number of specified virtual users (VUs) simultaneously started sending requests to the specified endpoint immediately. During tests with a higher number of VUs (depending on the node type/plan), the test concluded within 2.0 seconds. Based on these results, the decision was made to use a ramp-up time of 5.0 seconds. This ramp-up time allows for a gradual increase to the specified number of VUs over 5 seconds.

The impact of using a ramp-up time is also visible in the CPU usage graph generated by BTOP. As shown in Figure 9, the load on the CPU gradually increases before reaching its peak.

This adjustment to the testing methodology provided a more realistic simulation of how the system would handle increasing loads in a production environment. It also helped to better identify and measure the performance limits of each node plan while avoiding abrupt spikes that could skew results.
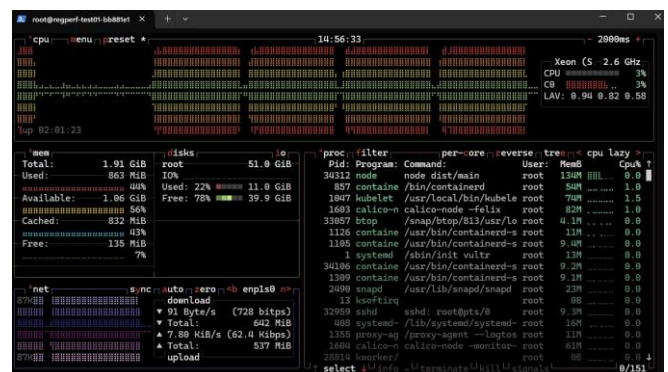

Figure 7: BTOP with ramp-up time.

Equal Number of Replicas as vCPUs?
In Kubernetes, it is possible to configure the number of replicas for an application, determining how many

instances of the application run concurrently. During the first round of tests, a node with one vCPU was used, initially running only one instance of the application. Follow-up tests, in which multiple instances were run on the same vCPU, revealed a significant performance degradation.

The same approach was tested on a node with multiple vCPUs. Results showed that a node with four vCPUs but only one instance managed significantly less load compared to configurations where the number of instances matched the number of vCPUs. This suggests that when the number of replicas matches the available vCPUs, overall performance improves.

This observation aligns with Kubernetes' core design principles, where scaling horizontally by increasing replicas improves load distribution. Running fewer replicas than the available vCPUs can lead to underutilization of resources, while running more replicas than vCPUs causes contention for CPU time, degrading performance.

A detailed analysis of these findings can be found in Appendix 6: Instance count vs. vCPU & memory usage**.**

Conclusion

Based on the tests conducted, it can be concluded that the selection of the most suitable node pool type (and node plan) at Vultr for scaling depends on the specific performance requirements and load demands of the application. The tests performed for the developed Content Management System demonstrate that the **High Performance (Intel)** node type delivers the best performance for both the client and API. When faced with high numbers of virtual users (VUs) and a high number of requests per second (RPS), this node type consistently outperforms others in the tests.

The third node plan of this type, which delivered the best results, is most suitable for scenarios where the application is under heavy load. However, based on an interview (see Appendix 3: Expected Number of Users), it is not expected that the application will experience such extreme load.

To remain cost-efficient, it is important to account for the expected load and then determine which node plan is best suited for scaling under typical usage scenarios. This approach ensures resources are used effectively without incurring unnecessary costs.

Reflection

While the chosen testing methodology can be considered robust, there are several limitations and areas for improvement that should be addressed in future tests.

Test Environment

A significant limitation is that not all tests were conducted in a fully controlled environment. This led to external factors that may have influenced the results, such as:

Network Limits

The test location had a shared internet connection with other devices and users. As a result, fluctuations in bandwidth and other network traffic could have impacted the test results, especially during intensive load testing of the client.

Download Speed

A specific limitation encountered was the 22 MB/s download speed of the Eduroam network. Tests at 2400 VUs, for instance, had to be performed using two devices, which likely underestimated the performance of high-capacity nodes.

Wi-Fi Usage

The test location only provided internet access via a Wi-Fi connection. A wired Ethernet connection might have yielded better results in terms of both speed and stability.

Hardware Limitations

By distributing the tests across multiple devices, additional risks were introduced, such as the limitations of the underlying infrastructure. This made the test setup more complex and less reliable, potentially affecting the precision of the results.

Physical Environment

The use of a shared network in a non-isolated space, along with factors like open applications and other active devices, may have contributed to variable performance during testing. Environmental distractions and competing demands for resources could have further skewed results.

Methode

Although we are confident in the test methodology chosen and executed, there is room for improvement, especially in handling edge cases and isolating issues.

Ramp up

One of the key findings was the decision to include a ramp-up period. During the first round of testing, a

ramp-up time of five seconds was chosen. However, this decision was made without any supporting analysis or justification. The assumption was that five seconds would be sufficient for the test to succeed, but there was no in-depth investigation into whether this duration was realistic or if it was too long or too short.

To improve future tests, it would be beneficial to experiment with different ramp-up durations to assess their impact on performance. Additionally, data analysis could be conducted to determine the optimal ramp-up time for each specific application and test condition.

Snapshots

Due to time constraints, the decision was made to run the tests consecutively rather than repeating them multiple times. In theory, this could mean that a specific node type might present different results at different times of the day.

In practice, this would likely have little to no noticeable impact on the measured values. However, this is an area that could be explored further in follow-up research, particularly to account for potential variations in system load, network conditions, and other factors that could affect the consistency of the results. Repeating tests at various times and under varying conditions would help ensure more reliable and accurate data.

Impact on Results

These factors have led to the possibility that the performance of the tested nodes may not fully represent a stable production environment. The influence of network fluctuations and physical limitations may have caused some tests to yield less reliable or inconsistent results.

Recommendations for Future Tests

To address these limitations and improve the reliability of future tests, the following actions are recommended:

Isolated Test Environment

Setting up a dedicated test environment, such as a local test lab or a cloud environment, could significantly improve the reliability of the measurements. At the very least, it is recommended to monitor and control the number of devices and applications that are open during testing.

Wired Network

Switching to an Ethernet connection would eliminate any fluctuations in Wi-Fi connections. This change could also bypass an additional device (the access point), potentially improving latency.

Higher Bandwidth

Testing over a connection with higher bandwidth can prevent download speed from becoming a limiting factor. This would be especially beneficial when testing clients or APIs that send large amounts of data back, as the higher bandwidth would better support these data transfers.

Multiple Testing Time Points

Repeating tests on different days and times can help eliminate performance variations and account for network traffic changes, ensuring more consistent and reliable results.

More Advanced Monitoring Tools

Using advanced monitoring tools such as Wireshark could make it easier to identify fluctuations in network speed or external influences during testing, providing deeper insights into any anomalies.

Additional Recommendations

To further improve the reliability of the analysis, all node types and plans could be tested in future research. This would provide a more comprehensive view of alternative options not covered in this study.

Additionally, as other node types were not tested in this research, it may be worth considering them in the future, especially if visitor numbers grow significantly, as they may offer viable alternatives.

Conclusion

Despite the limitations, we stand by the findings of this research. The chosen test methodology, which included thresholds and a step-by-step approach, was carefully designed to produce reproducible and reliable results. Repeating tests to identify edge cases and organizing result documentation in an Excel file further contributed to the reliability of the test method.

We are confident that the results provide an accurate representation of the performance of the tested node types and configurations. For future work, it would be valuable to create a fully isolated test environment and use more advanced monitoring tools to further minimize external influences.

Bibliografie

Ahmedov, J. (2018, 22 7). *How does an elevator works?* Retrieved from dev.to: https://dev.to/jimjja/how-does-an-elevator-works-1g40#:~:text=All%20algorithms%20might%20differ%20from,known%20as%20I%2FO%20scheduling.

B.P.Pokharel. (2021). *A Comparative Analysis of Disk Scheduling Algorithms.* International Journal for Research in Applied Sciences and Biotechnology.

Bansal, N. (2024, Mei 6). *An Overview of Caching for PostgreSQL*. Retrieved from Severalnines: https://severalnines.com/blog/overview-caching-postgresql/

Datacenter.com. (2024, 12 16). *What is a vCPU and How Do You Calculate vCPU to CPU?* Retrieved from Datacenter.com: https://www.datacenters.com/news/what-is-a-vcpu-and-how-do-you-calculate-vcpu-to-cpu

GeeksForGeeks. (2024, April 17). *https://www.geeksforgeeks.org/introduction-postman-api-development/*. Retrieved from GeeksForGeeks: https://www.geeksforgeeks.org/introduction-postman-api-development/

Grafana. (2024, 12 13). *Grafana k6*. Retrieved from Grafana: https://grafana.com/docs/k6/latest/

Hayhurst, D. (2022, Juli 6). *How much RAM does NextJS use?* Retrieved from Cotemplates: https://cotemplates.com/nextjs-templates/how-much-ram-does-nextjs-use/

Labs, G. (2024). *K6*. Retrieved from https://k6.io/open-source/

Node.js. (2024, 12 13). *Node.js*. Retrieved from Node.js v23.4.0 documentation: https://nodejs.org/api/perf_hooks.html

Ubuntu. (2024, 12 16). *System requirements*. Retrieved from Ubuntu: https://ubuntu.com/server/docs/system-requirements

Windesheim. (2023). *Over Windesheim*. Retrieved from Windesheim: https://www.windesheim.nl/over-windesheim/zwolle

Appendix

Appendix 1: The assignment

Quality in Software Development

Registration Form for Company Assignments

This form is used for:
- Screening assignments by Windesheim, HBO-ICT
- Distributing assignments among QSD students so they can contact clients as interested parties.

Please at least fill in the dotted lines. The form may always be supplemented with more information.

Please read the document "QSD Project – Guidelines and Criteria for Company Assignments."

Company details

Company	Startup
Location	Zwolle
Project	Horizontal Scaling Insights
Contact Person	

Project Definition

What does the company do? What challenges does the company face?

The startup is still in the startup phase, working on developing a web app. It has been decided to deploy the web app on a Kubernetes cluster with a microservice architecture. The advantage of this architecture is the ability to scale horizontally, meaning offering multiple containers for the same application. When scaling up, more resources are used, which leads to higher costs. It is not entirely clear which cloud computing specifications available from the provider, in combination with the traffic volume, are the most suitable. So, the question is what the correct threshold values are in combination with the selected settings for the cluster.

Assignment Description

What is seen as the solution(s)?

The proposed solution consists of several parts. The first step is to map out the relevant factors. Based on research, it should become clear what might work. Next, a Proof of Concept (POC) will be built. The techniques used in the POC should align with the existing technologies, so the project can be used in production.

The techniques used include:
- A frontend with NextJS
- Microservices with NestJS
- Deployment with Helm on a Kubernetes cluster
- Logging with Prometheus and Grafana
- Optionally, the use of Cloudflare products such as R2 or Stream
- By executing the outcomes of the research on the POC, it should become clear whether the scaling of the cluster and the use of resources is efficient and transparent.

Scope

Is there enough work for 5-6 QSD students for 10 weeks?

It is a challenging and extensive project with a high degree of depth.

Where is the technical complexity? What is the challenge?

The technical complexity lies in:
- The project takes place in the areas of software and infrastructure.
- The project uses (cor)relation of metrics from VMs, pods, and containers.
- The use of Kubernetes in combination with a microservice architecture
- The project contains several technical components that will be developed in parallel and must align closely at the same time.

What supervision can the company provide?

Weekly guidance and meetings.

Where is the technical complexity? What is the challenge?

The technical complexity lies in:

Appendix 2: API-query times

To obtain a realistic view of the client, the API is simulated. This is done so that the speed of the API does not affect the speed of the client. An element that should be included in the page load time, however, is the loading time of the content. Retrieving HTML, CSS, and JavaScript takes only a few milliseconds. Fetching the page content, such as content items, takes longer. To still get a realistic view of the load time, the average load time of an API call is considered.

API-endpoints

Table 12 provides an overview of the tested API endpoints.

Endpoint id	URL	Type	Request format	Parameters	Return value
1	/auth/login	POST	503 bytes	Username, password	JWT-auth token
2	/content/get-range	POST	6807 bytes	RangeMin, RangeMax	JSON-object met content

Table 12: Tested API Endpoints

Test Devices

Two different devices were chosen to account for differences in hardware performance, as the device speed can influence the results. This provides a broader overview of how the API endpoints perform under different conditions.

Device id	Device Name	CPU	RAM
1	Martijn's Desktop	AMD Ryzen 7 7800x3D	32GB DDR5 6000mhz
2	Martijn's Laptop	Intel i5 1135g7	20GB DDR4 3200mhz

Table 13: Devices used for the tests.

Test Conditions

The API runs on a local Kubernetes cluster and uses a simulated database. This setup allows the API to be tested fully isolated from external systems. Each test uses the same parameters. The tests were conducted using Postman, which makes it easy to test API endpoints (GeeksForGeeks, 2024).

Test Parameters

Different parameters are required for each API endpoint. The parameters used are shown in Table 14.

Endpoint	Parameter 1	Parameter 2
/auth/login	TestUser	TestPassword
/content/get-range	0	10

Table 14: Parameters used for API calls.

Test Results

Table 15 displays the results of the conducted tests.

#	Endpoint	Test device	Reaction time (ms)
1	1	1	5
2	1	1	3
3	1	1	3
4	1	1	3
5	1	1	6
6	1	1	3
7	1	1	3
8	1	1	3
9	1	1	3
10	1	1	3
11	1	1	4
12	1	1	3
13	1	1	3
14	1	1	3
15	1	1	3
16	1	1	4
17	1	1	3

18	1	1	3
19	1	1	3
20	1	1	4
21	2	1	4
22	2	1	3
23	2	1	3
24	2	1	4
25	2	1	4
26	2	1	4
27	2	1	4
28	2	1	3
29	2	1	3
30	2	1	4
31	2	1	4
32	2	1	4
33	2	1	3
34	2	1	5
35	2	1	3
36	2	1	2
37	2	1	3
38	2	1	3
39	2	1	4
40	2	1	3
41	1	2	43
42	1	2	16
43	1	2	17
44	1	2	16
45	1	2	15
46	1	2	16
47	1	2	15
48	1	2	15
49	1	2	16
50	1	2	17
51	1	2	15
52	1	2	15
53	1	2	15
54	1	2	17
55	1	2	33
56	1	2	16
57	1	2	14
58	1	2	15
59	1	2	29
60	1	2	24
61	2	2	33
62	2	2	15
63	2	2	16
64	2	2	18
65	2	2	18
66	2	2	18
67	2	2	19
68	2	2	22
69	2	2	24
70	2	2	16
71	2	2	17
72	2	2	18
73	2	2	16
74	2	2	18

75	2	2	17
76	2	2	24
77	2	2	18
78	2	2	16
79	2	2	20
80	2	2	19

Table 15: Results of the tests

Average Response Times

The average response times between different devices and the same API endpoints vary minimally. This is likely due to hardware differences. The difference in response times between the different API endpoints is partly related to the amount of data transmitted, with 503 bytes compared to 6807 bytes.

API Endpoint	Device	Average Response Time (ms)
1	1	3.40
1	2	3.50
2	1	18.95
2	2	19.10

Table 16: Average Response Times

Ping to Vultr

To get a better picture of the total response time, the server's ping was also measured. This was done by performing a ping from both test devices. The ping was measured on two days at three different times of the day, each over a span of 60 seconds. The times were 10:00, 15:00, and 20:00. The node being pinged at Vultr was a regular 2 vCPU 2GB machine hosted in Amsterdam.

The results of these measurements are shown in Table 17.

Day	Time	Device	Average Ping after 60 seconds (ms)
1	10:00	1	5.1
1	10:00	2	5.4
1	15:00	1	5.0
1	15:00	2	5.0
1	20:00	1	5.2
1	20:00	2	5.3
2	10:00	1	5.0
2	10:00	2	5.2
2	15:00	1	6.1
2	15:00	2	5.4
2	20:00	1	5.1
2	20:00	2	5.3

Table 17: Results of Ping Tests

Average: 5.26ms

Conclusion

From the measured response times, it can be concluded that the average API response time ranges from 3.4 to 19.10 milliseconds, depending on the endpoint and the test device. When the average ping to Vultr is added, the total average response time ranges from 8.66 to 24ms.

However, in a production environment, average response times and ping are not static. Elements such as load on the infrastructure, network configurations, and device performance can cause changes. To account for this in the simulation, a small delay was added. With this correction, the response time is rounded to 30ms, providing a realistic baseline for further simulations and tests.

Appendix 3: Database query timing

The following appendix contains the results of tests conducted on the PostgreSQL database, where different SQL queries were tested to measure the response and processing times. These data are used to support the choice of delay time in the prototype.

Query types

Table 18 provides an overview of the queries used. Both queries retrieve information for the "My Channel" page (UC2). The first query retrieves data for page 1, while the second query retrieves data for page 2. The difference between the queries lies in the OFFSET 10.

Query id	Query inhoud
1	SELECT "content"."id" AS "content_id", "content"."title" AS "content_title", "content"."publicationStatus" AS "content_publicationStatus", "content"."type" AS "content_type", "content"."contentItemID" AS "content_contentItemID", "content"."createdAt" AS "content_createdAt", "content"."updatedAt" AS "content_updatedAt", "content"."deletedAt" AS "content_deletedAt", "content"."userIDId" AS "content_userIDId" FROM "content" "content" WHERE "content"."userIDId" = $1 ORDER BY "content"."createdAt" DESC LIMIT 10
2	SELECT "content"."id" AS "content_id", "content"."title" AS "content_title", "content"."publicationStatus" AS "content_publicationStatus", "content"."type" AS "content_type", "content"."contentItemID" AS "content_contentItemID", "content"."createdAt" AS "content_createdAt", "content"."updatedAt" AS "content_updatedAt", "content"."deletedAt" AS "content_deletedAt", "content"."userIDId" AS "content_userIDId" FROM "content" "content" WHERE "content"."userIDId" = $1 ORDER BY "content"."createdAt" DESC LIMIT 10 OFFSET 10

Table 18: Overview of Used Queries

Test Devices

Two different devices were chosen to account for hardware performance differences, as the speed of the devices can influence the results. This provides a broader overview of how the queries perform under varying conditions.

Device ID	Device Name	CPU	RAM
1	Martijn's Desktop	AMD Ryzen 7 7800x3D	32GB DDR5 6000mhz
2	Martijn's Laptop	Intel i5 1135g7	20GB DDR4 3200mhz

Table 19: Devices Used for Testing

Test Conditions

The database was recreated and filled with test data before each test to ensure the consistency of results. This was done to prevent earlier query executions from influencing the test results, as PostgreSQL uses caching, which can affect response times (Bansal, 2024).

Each test was measured using the Node.js Performance Hook (Node.js, 2024). This allows us to log the time taken by a query to execute. The logging is carried out as follows:

```
// Start time
const startTime = performance.now();

// Fetch contents with pagination
const contents = await this.contentRepository.find({
  where: { userID: userId },
  order: { createdAt: 'DESC' },
  take: rangeSize,
  skip: rangeMin,
});

// End time
const endTime = performance.now();
console.log(`get-all DB response: ${endTime - startTime}ms`);
```
Figure 8: Code Example for Logging Query Response Time

In the code shown in Figure 8, the start time is recorded before the query is executed. Once the data is retrieved, the end time is stored. The difference between the start and end time is then logged. This difference represents the time the query took to execute.

Test Results

Table 20 shows the results of the tests. The effect of PostgreSQL caching is clearly visible, especially when comparing the results of tests 1, 21, 41, and 61 with the subsequent tests.

#	Query	Test device	Reaction time(ms)
1	1	1	6,4352
2	1	1	1,7120
3	1	1	1,3953
4	1	1	1,4169
5	1	1	1,1034
6	1	1	1,5287
7	1	1	1,0567
8	1	1	1,5614
9	1	1	1,1418
10	1	1	1,4059
11	1	1	1,1626
12	1	1	1,1852
13	1	1	1,8470
14	1	1	1,6931
15	1	1	1,6244
16	1	1	1,8151
17	1	1	1,1921
18	1	1	0,9566
19	1	1	1,5238
20	1	1	2,98766
21	1	2	10,05134
22	1	2	3,792972
23	1	2	1,905924
24	1	2	2,535176
25	1	2	1,826438
26	1	2	0,795047
27	1	2	1,903121
28	1	2	1,880441
29	1	2	0,677074
30	1	2	0,966969
31	1	2	1,152637
32	1	2	0,789235
33	1	2	2,81304
34	1	2	1,600596
35	1	2	2,497985
36	1	2	1,003609

37	1	2	0,811887
38	1	2	1,657102
39	1	2	1,099705
40	1	2	1,145002
41	2	1	4,945876
42	2	1	2,452308
43	2	1	1,536457
44	2	1	1,006479
45	2	1	2,176118
46	2	1	2,999659
47	2	1	4,754347
48	2	1	3,876479
49	2	1	1,869832
50	2	1	1,836627
51	2	1	1,512322
52	2	1	1,921616
53	2	1	1,574164
54	2	1	0,797591
55	2	1	2,94279
56	2	1	1,527796
57	2	1	1,754103
58	2	1	1,157909
59	2	1	2,607892
60	2	1	0,794401
61	2	2	6,481943
62	2	2	3,331115
63	2	2	1,993493
64	2	2	1,557193
65	2	2	0,849311
66	2	2	1,495643
67	2	2	1,106535
68	2	2	1,697452
69	2	2	0,909893
70	2	2	0,423332
71	2	2	1,510593
72	2	2	1,253932
73	2	2	2,094692
74	2	2	4,461046
75	2	2	1,271103
76	2	2	1,493077
77	2	2	1,396868
78	2	2	2,502137
79	2	2	1,881649
80	2	2	1,594401

Table 20: Results of the carried-out tests.

Average Response Times

The average response times between the different devices and queries differ slightly. This is likely due to hardware differences as well as the varying types of queries. The higher values for 'Martijn's Laptop' may stem from the fact that the hardware specifications of this device are lower than those of 'Martijn's Desktop'. This is further supported by comparing the specifications of both devices.

Query	Device	Average Response Time (ms)
1	1	1.73
1	2	2.04
2	1	2.20
2	2	1.96

Table 21: Average Response Times

Conclusion

Based on the average response times, it can be concluded that PostgreSQL uses caching, which results in significantly lower response times after the first request. The average response time ranges from 1.73 ms to 2.20ms, depending on the query and the test device.

In a production environment, these response times would likely be higher due to factors such as server load, concurrent users, and load on the infrastructure. This aspect was not simulated in this test but would play a role in a live environment. Additionally, the test results show that hardware performance has a measurable impact: the laptop with lower specifications exhibits higher response times compared to the desktop.

The performance differences between the queries demonstrate that more complex SQL constructs, such as using OFFSET, can cause a slight increase in response time.

Appendix 4: K6 example

In Figure 9 and Figure 10, a load test is performed using 700 virtual users configured with a ramp-up period of five seconds. During the ramp-up phase, the number of virtual users increases gradually from 0 to 700 over five seconds, after which the full server load is applied. This gradual approach ensures that the server is incrementally stressed, allowing for reliable and realistic test results.
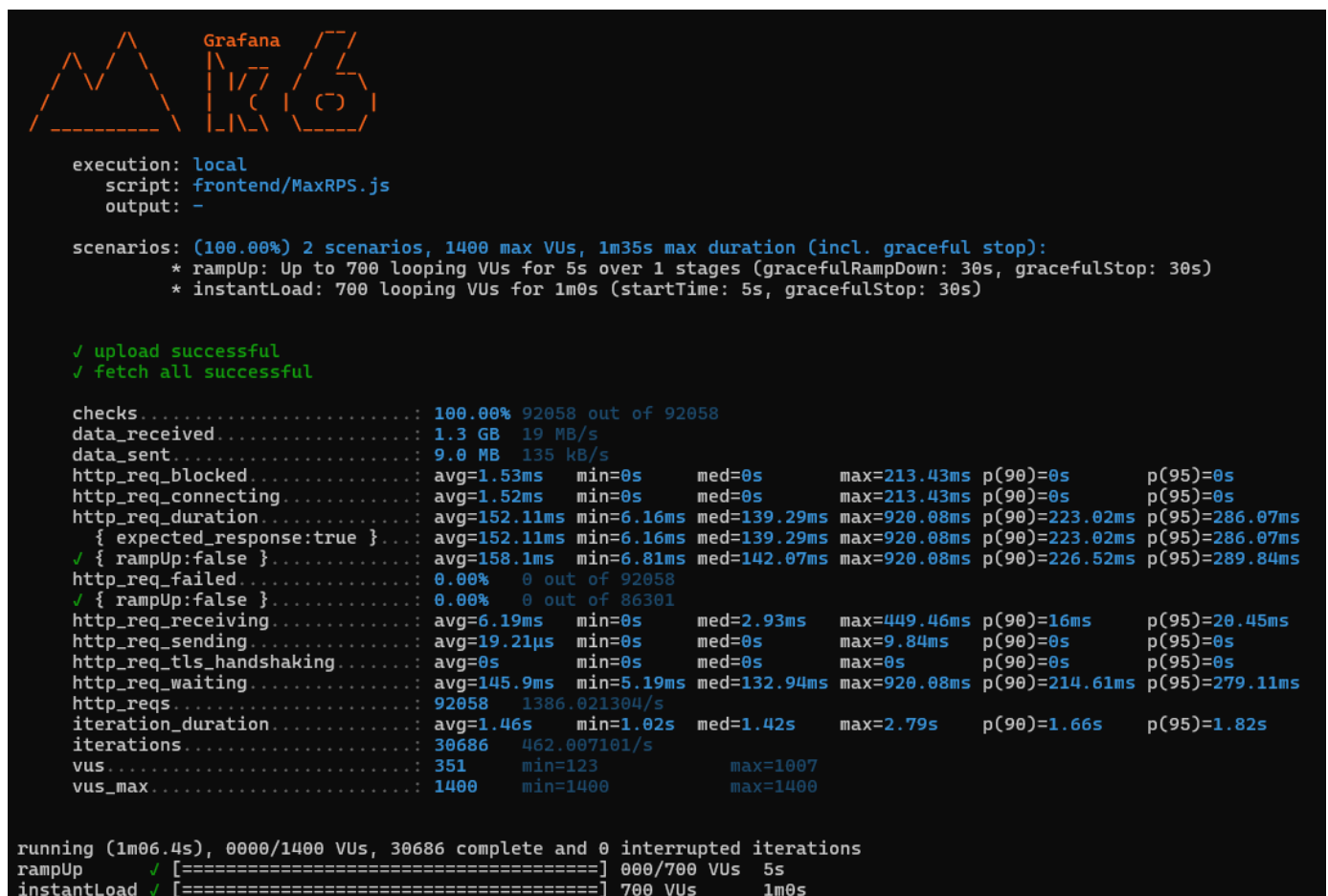
Figure 9: K6 test result.
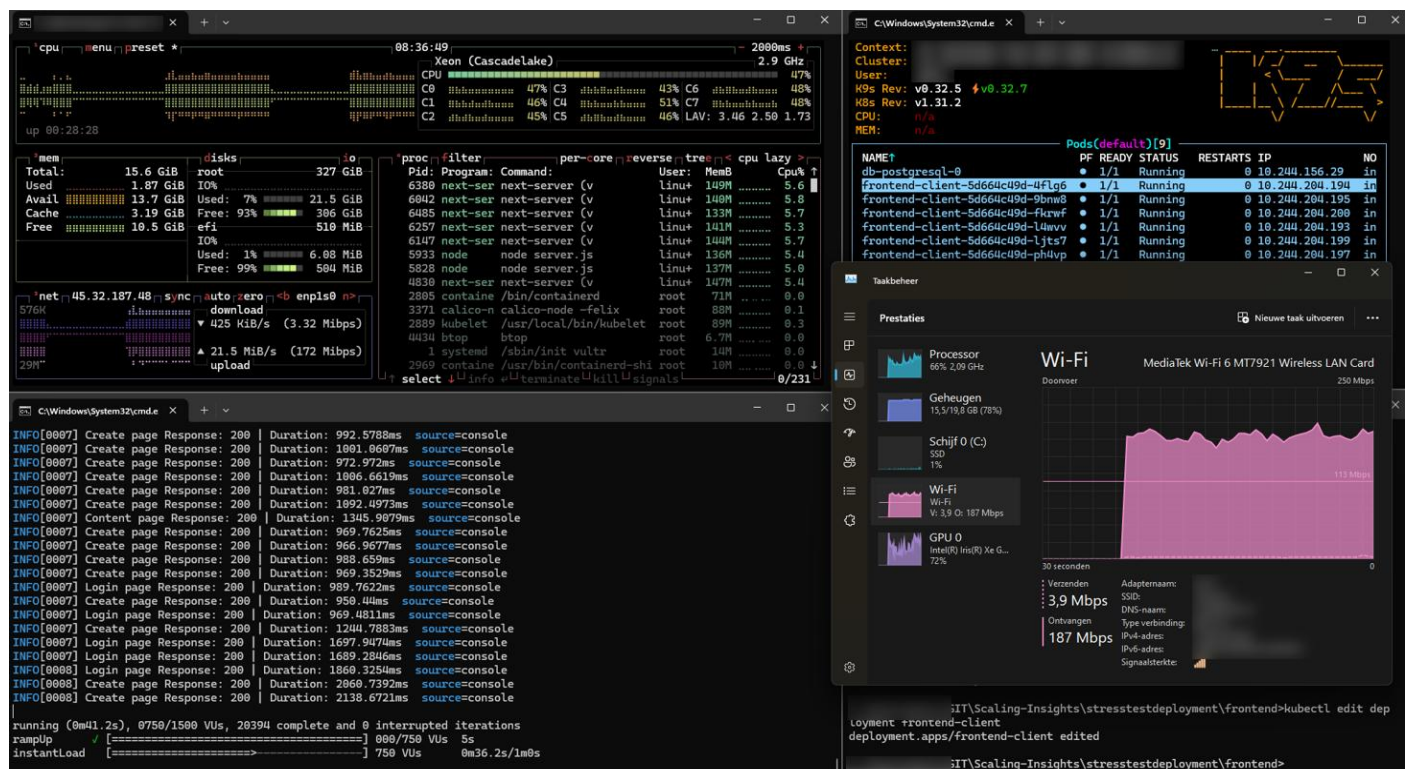
Figure 10 Graphs and metrics from BTOP, K9, K6, and network traffic during node load testing.

After the ramp-up phase, an **instantLoad test** begins, which lasts for one minute. During this test, all 700 VUs send parallel HTTP requests to the endpoints specified in the configuration. This simulates peak load conditions and measures the system's performance under high demand.

The table in **Figure 2** shows the results of the load test:

- **100% of checks passed**, demonstrating no errors during the test.

- A total of **92,058 requests** were successfully sent.

- The **median response time** was 139.29ms, with the fastest response being 6.16ms and the slowest 920.08ms.

- Importantly, **no requests were dropped or terminated by the server**, confirming the system's robustness under load.

This test is considered successful as it meets all predefined thresholds, including 0% request loss and a 95th percentile response time under 1000ms. These results provide confidence in the system's ability to manage high volumes of traffic effectively.

By gradually increasing the load and simulating real-world usage patterns, K6 ensures that the results are both accurate and actionable. For a deeper analysis of the test setup and performance metrics, see the graphs and additional details in **Figure 3**.

Appendix 5: Expected users.

To limit the number of node types to be tested, an estimate of the expected number of users for the application was made in consultation with the client. This number serves as an important basis for selecting appropriate node types, ensuring the research is carried out efficiently.

Relevance of Node Types

Based on the expected number of users, it can be concluded that the application will generate average to moderate load. The following assumptions were made:

Optimized Cloud Compute Node Types Are Not Needed:
The more expensive Optimized Cloud Compute node types, such as those with a large number of vCPUs or a high amount of RAM, are intended for heavy loads or specialized applications. Since these are far beyond the expected usage, they are not relevant for this study.

Nodes with Insufficient Capacity Are Excluded:
Nodes with less than 1GB of RAM or a limited number of vCPUs do not meet the minimum requirements to run the application stably, even with a small number of concurrent users. Nodes using Ubuntu Server are considered, as Ubuntu recommends at least 1GB of RAM for the server(Ubuntu, 2024).

During testing, a question arose regarding the expected number of users on the application. This question was posed to the client (Ernst Bolt).

Client Discussion
Martijn Schuman: "How many users do you expect?"
Ernst Bolt: "200 VUs with 2 RPS... Worst case 400 VUs with 4 RPS (without preflight), then it would be 8 RPS."

Appendix 6: Instance count vs. vCPU & memory usage

Within the Vultr and Kubernetes infrastructure, it is possible to configure nodes and instances in various ways. For example, a single vCPU core can be allocated to multiple instances, which could theoretically lead to more efficient resource utilization.

There were questions about the optimal balance:
- Is it more efficient to split a vCPU across multiple instances?
- Or does it provide better performance to allocate a single instance per vCPU?

To answer these questions, tests were conducted to explore the relationship between the number of vCPUs and instances. The focus was specifically on the following:
- Average CPU usage per vCPU and per instance.
- RAM usage per instance.
- Maximum load that a node can handle, expressed in the number of Virtual Users (VUs).

Predictions:
Before conducting the tests, two predictions were made:

1. **Limited Multithreading of Next.js:** Since Next.js is not fully multithreaded, it might be difficult to utilize multiple vCPUs effectively. This could result in suboptimal performance when configurations involve multiple instances per vCPU.
2. **Inefficient Handling of Short CPU Spikes:** Live CPU monitoring suggests that vCPUs struggle with short bursts of load. With configurations that involve multiple instances per vCPU, this could lead to overload on individual cores, negatively impacting overall performance.

To test these predictions and understand the efficiency of various configurations, multiple tests were performed on nodes with different combinations of vCPUs and instances. The test results provide insights into the efficiency of different configurations and help determine which settings are most suitable for scalable applications.

Test Setup:
The same test script was used for all tests, ensuring consistent settings. The client was used for testing.

The key test parameters are as follows:
- **Virtual Users (VUs):** The number of concurrent users was gradually increased until the maximum node load was reached.
- **CPU Usage**: The average CPU usage per vCPU and per instance was tracked during the tests.
- **RAM Usage**: The average memory usage per instance was measured.
- **Test Repetitions**: Each configuration was tested four times to ensure consistent results.

Tests were performed on nodes with one and four vCPUs, with the number of instances per vCPU varying from one to eight. The maximum load (VUs) was determined by stressing the node until the test thresholds were exceeded.

To collect server load data, a test script called K10 was developed. This script was installed directly on the virtual machine and gathered server metrics over the test duration. Documentation for K10 can be found in Appendix 9: K10.

Results
Tables 22 and Figure 11 present the results of the tests conducted. The columns "Average vCPU Usage", "Average vCPU Usage per Instance", "Average Total RAM Usage", and "Average RAM Usage per Instance" are structured as follows:

Horizontal Axis
The horizontal axis of each column displays the different test results. For instance, for the node with one vCPU and one instance, the first test showed an average vCPU usage of 84%, the second test 85%, the third test 84%, and the fourth test 83%. These results are displayed side by side in the same row.

Vertical Axis (Average vCPU Usage)

For the "Average vCPU Usage" column, the vertical axis is based on the number of vCPUs. Nodes with one vCPU, like in the first four tests, have one row on the vertical axis. Nodes with four vCPUs, like in the later tests, have four rows, with the usage per individual vCPU being recorded.

Vertical Axis (Average vCPU Usage per Instance & Average RAM Usage per Instance)
For the "Average vCPU Usage per Instance" and "Average RAM Usage per Instance" columns, the vertical axis is based on the number of instances. A test with one instance has one row on the vertical axis, while a test with four instances has four rows. The average vCPU usage and RAM usage per instance are displayed separately for each instance.

This approach ensures clear representation of how different configurations (number of vCPUs and instances) impact CPU and memory usage in terms of both aggregate and per-instance values.

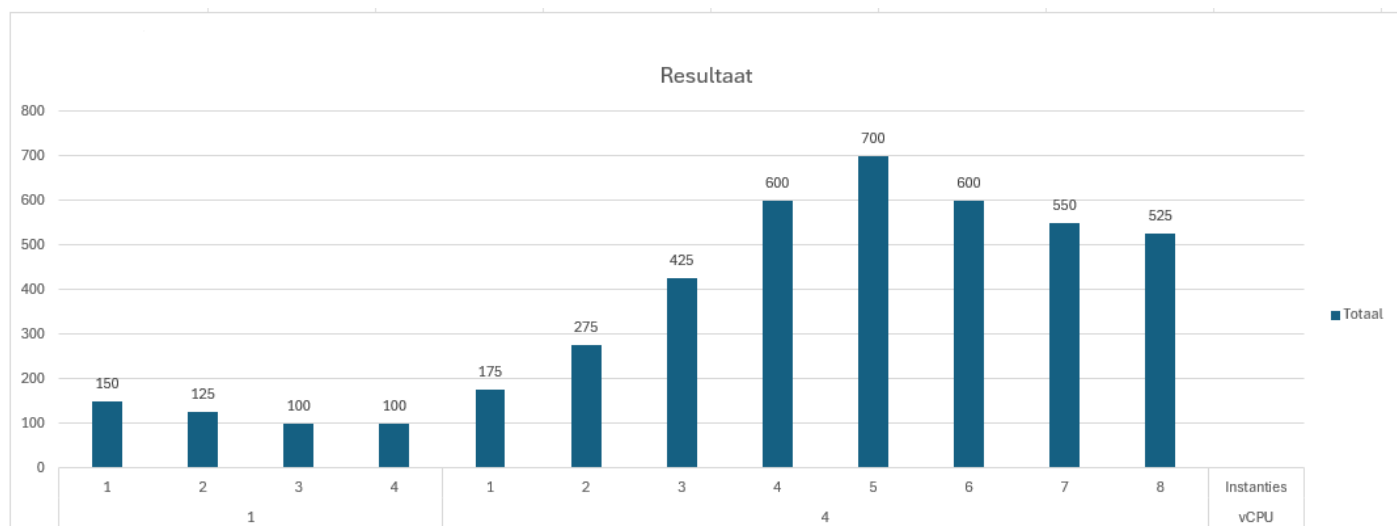| vCPU	Instances	Average vCPU-usage (%)	Average vCPU-usage per instance (%)	Average total RAM-usage (MB)	Average RAM-usage per instance (MB)	Max VUs
1	1	0: 84 85 84 83	0: 83 84 84 83	118 121 121 101	0: 118 121 121 101	150
1	2	0: 84 83 83 82	0: 42 41 42 42 1: 42 43 41 42	241 240 238 236	0: 121 120 120 119 1: 120 120 118 117	125
1	3	0: 80 78 78 78	0: 29 27 30 30 1: 25 24 21 21 2: 26 28 27 27	350 349 349 347	0: 117 116 117 116 1: 116 114 115 113 2: 118 118 117 117	100
1	4	0: 82 78 80 82	0: 19 15 21 21 1: 19 24 21 20 2: 22 17 18 22 3: 22 22 19 18	460 459 461 460	0: 115 115 118 115 1: 114 115 115 115 2: 117 115 115 115 3: 113 114 113 115	100
4	1	0: 27 23 17 27 1: 28 27 28 26 2: 22 28 27 23 3: 26 19 24 22	0: 26 24 23 25	170 174 183 177	0: 170 174 183 177	175
4	2	0: 39 41 48 51 1: 56 56 46 45 2: 44 42 54 50 3: 50 50 45 48	0: 96 97 100 98 1: 99 97 98 101	349 371 347 344	0: 178 188 175 172 1: 171 183 172 172	275
4	3	0: 69 69 67 64 1: 64 66 69 69 2: 64 64 64 65 3: 70 63 70 67	0: 94 94 92 93 1: 93 95 96 94 2: 94 88 94 92	552 532 529 512	0: 187 184 183 168 1: 176 168 170 172 2: 189 180 176 173	425
4	4	0: 82 82 83 83 1: 84 84 82 83 2: 83 84 83 84 3: 86 85 82 83	0: 88 90 86 92 1: 91 90 90 92 2: 90 89 93 91 3: 91 92 90 85	726 723 674 694	0: 174 183 171 182 1: 187 186 170 176 2: 178 180 169 173 3: 187 173 165 163	600
4	5	0: 85 86 86 86 1: 85 87 88 88 2: 86 84 83 83 3: 85 85 86 86	0: 73 78 78 73 1: 70 74 75 74 2: 76 75 75 75 3: 72 74 74 74 4: 74 70 70 74	831 895 871 873	0: 168 174 175 171 1: 167 174 173 172 2: 171 192 168 177 3: 159 175 175 178 4: 166 180 180 175	700
4	6	0: 87 87 86 85 1: 86 86 85 85 2: 86 86 83 84 3: 86 86 89 90	0: 62 62 59 62 1: 61 61 63 62 2: 64 64 64 65 3: 61 64 62 55 4: 61 60 55 59 5: 63 61 65 63	1029 1030 1001 1046	0: 171 171 168 170 1: 161 168 166 179 2: 177 174 168 175 3: 176 176 173 174 4: 174 173 164 171 5: 171 167 162 177	600
4	7	0: 82 84 80 86 1: 83 85 80 86 2: 85 84 81 85 3: 79 83 80 85	0: 45 50 57 57 1: 46 54 37 52 2: 49 49 67 53 3: 53 50 45 49 4: 47 59 63 54 5: 60 54 42 50 6: 50 40 30 51	1099 1042 1076 1094	0: 153 155 151 159 1: 159 148 160 159 2: 157 144 166 155 3: 150 155 146 170 4: 149 150 159 156 5: 169 150 163 159 6: 163 140 132 136	550
4	8	0: 86 82 83 85 1: 83 86 86 86 2: 86 85 86 82 3: 88 87 88 87	0: 50 48 46 43 1: 51 49 49 48 2: 47 49 48 47 3: 54 41 50 45 4: 47 49 38 47 5: 48 46 44 44 6: 34 41 48 47 7: 33 44 38 43	1118 1253 1231 1269	0: 156 160 147 161 1: 153 156 155 159 2: 148 163 163 168 3: 157 159 155 163 4: 148 159 147 158 5: 149 163 156 148 6: 136 148 153 163 7: 141 145 155 148	525

Table 22: Results of the carried-out tests.

Figure 11: Results of the carried-out tests.

Interpretation of Test Results

CPU Usage:

- As the number of instances per vCPU increases, the average vCPU usage per instance decreases. This suggests that splitting a vCPU across multiple instances leads to inefficient use of CPU resources.
- Configurations with one instance per vCPU perform more consistently and achieve higher maximum load (VUs).
- In some measurements, the total value in Table 22 for "Average vCPU Usage per Instance" does not match the value in the "Average vCPU Usage" column. This is because "Average vCPU Usage" considers total usage, not just that of the Next.js server. The operating system and other system processes are also included here.

Maximum Load:

- Nodes with more instances per vCPU perform worse under higher load. The maximum VUs decrease as more instances are assigned to the same vCPU.
- Configurations with four vCPUs and four instances deliver the best performance, with a maximum load of 600 VUs. When eight instances per vCPU are used, the maximum load drops to 525 VUs, despite higher resource utilization [1].

RAM Usage:

- RAM usage remains constant per instance, regardless of the number of instances per vCPU. This indicates that RAM usage is not a direct limiting factor in these tests.
- However, when testing with more instances, RAM usage is slightly higher, which can be explained by the increased number of VUs that each instance must manage.

Predictions Confirmed:

- **Multithreading Limitations in Next.js:** The results support the hypothesis that Next.js does not effectively utilize multithreading. As a result, the potential of multiple vCPUs is not fully leveraged.
- **Impact of CPU Peaks:** Impact of CPU Peaks: Monitoring shows that short CPU peaks are poorly managed in configurations with multiple instances per vCPU. This leads to inefficiencies when processing concurrent load. See Figure 12: vCPU Usage During Testing.

Figuur 1: vCPU usages during testing.

---

[1] The test of five instances on four vCPUs is an outlier, there is no logical explanation or substantive analysis done as to why it performs better.

Conclusion

The tests demonstrate that the performance of Kubernetes nodes is highly dependent on the balance between vCPUs and instances.

The key findings are:
1. **One instance per vCPU**: Configurations with one instance per vCPU deliver consistently better performance and higher maximum loads.
2. **Splitting vCPUs is ineffective**: Splitting vCPUs across multiple instances leads to inefficient resource usage and lower performance, especially under high load.
3. **Recommendation**: For optimal scalability, it is recommended to keep the number of instances equal to the number of vCPUs. This prevents inefficient resource utilization and minimizes the impact of short CPU peaks.

These results provide a valuable foundation for future configuration decisions within Kubernetes clusters and highlight the need for further optimization of multithreading in applications like Next.js.

Appendix 7: Explanation test situation

Client

In order to simulate the backend during client testing, the method that typically performs HTTP-POST actions was modified. This method now always returns the same data. This allows for testing of the client's performance without depending on a working backend.

Additionally, network latencies were taken into account. A delay of 30 milliseconds was added to each HTTP request to simulate realistic latency. This delay was chosen based on average values and is further explained in Appendix 2: API-query times.

Cloudflare simulation was achieved by using local files to fetch thumbnails. Since the fetching of images is asynchronous, no additional delay was added for this action. This approach ensures an accurate simulation of the actual client flow, with a simulated backend and Cloudflare.
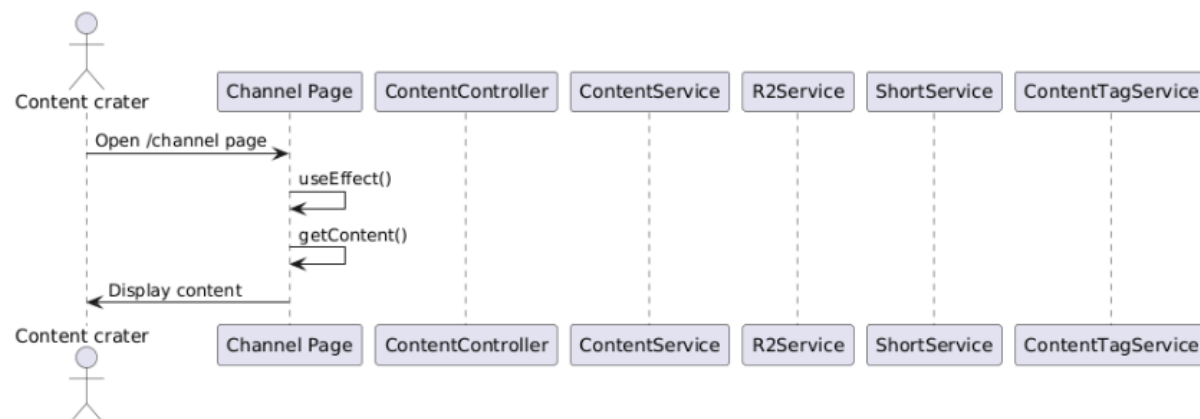

Figure 12: Flow for fetching with the simulated backend.

Backend

For backend testing, the database was simulated using a MockContentRepository. This repository is designed to always return the same data for read operations and to handle write or update actions without real interactions with a database.

The decision to simulate the backend with a MockContentRepository was deliberately made to minimize the need for changes to the backend code. This ensures that the data flow through the existing controllers and associated services remains intact, increasing the validity of the tests.

This approach allowed the performance of the backend to be evaluated independently, without the results being affected by any limitations or delays in the database.
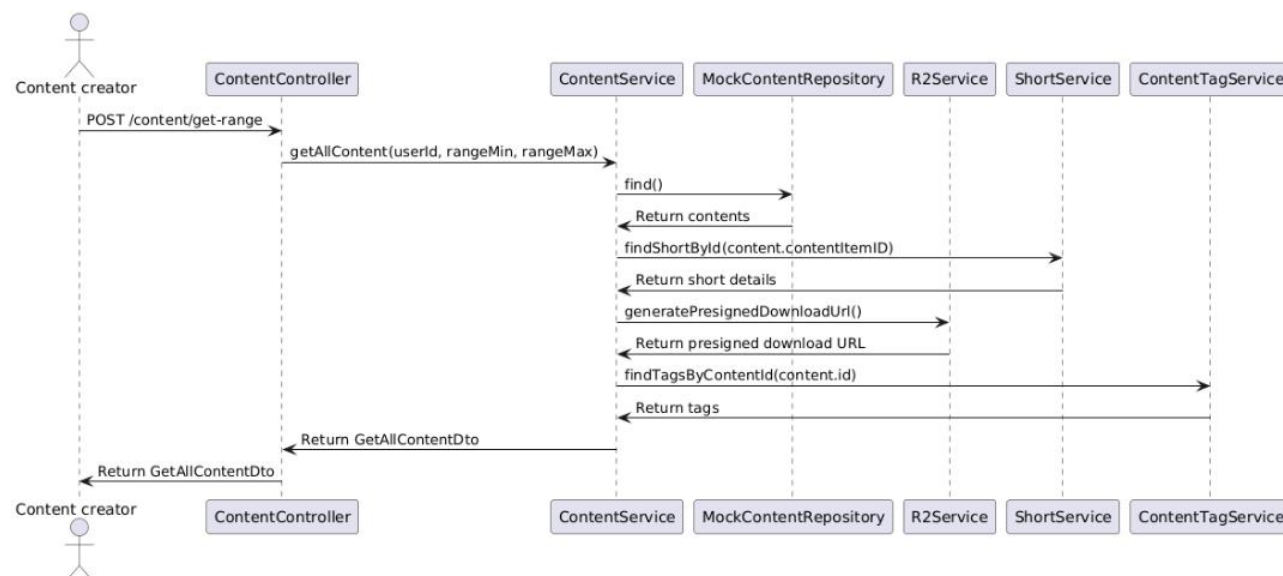

Figure 13: Flow for fetching with the simulated database.

Appendix 8: K7

The execution of all load tests in this research is a time-consuming process that requires two people to continuously monitor the process to start the tests and record the results. To simplify this process for possible follow-up research or reproduction, a script called K7 has been written. This appendix describes the functionality and design of K7. For a step-by-step guide on how to use K7, please refer to the read.me file on the [GitHub repository.](#)

K7 is a Python CLI (Command-Line Interface) application that automatically runs multiple K6 tests on the same endpoints. K7 continues testing until it determines the maximum number of virtual users that the server, the testing device, or the network can handle based on the provided values. It is possible to adjust several variables for a K7 test.

Variable Arguments

Several variable values can be passed to a K7 test:

- The duration of a K6 test.
- The time interval between K6 tests.
- The ramp-up time (time to gradually increase the number of virtual users).
- The initial number of virtual users.
- The increase in the number of virtual users.
- The number of attempts.
- The number of validations.

In addition, there is an argument for full logging and one for adjusting the location of the K6 script.

Reproducing the Research

To run the same tests as those in this research, you can execute the following command:

```
python k7.py --increment 100 --validation_runs 3 --delay_between_tests 5 --duration 60 --rampup_time 5 --fails 1
```

When this command is executed, the program will prompt for the initial number of virtual users, which can be adjusted based on the expected outcome to limit the test duration.
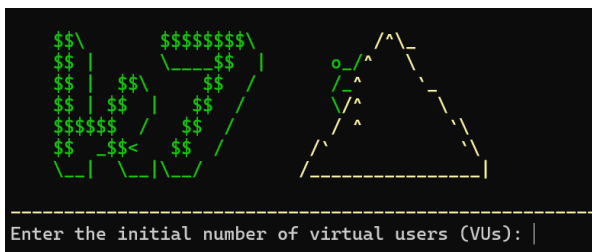

Figure 14: K7 Start Screen

Once this is filled in and the K6 test script is correctly configured, K7 will determine the maximum number of virtual users that the specified system can handle.

Figure 15: K7 Test Final Result

To reproduce this research, a K7 test should be conducted once per node plan for each application running on the cluster. The results of the K7 tests can then be compared to identify the best node plans for each application.

Appendix 9: K10

K10 is a custom-built monitoring script designed to track CPU and RAM usage per vCPU and per core. This appendix describes the design, functionality, and key decisions made during development.

Key Features

CPU Monitoring per Core:
- Uses mpstat to monitor CPU usage per core at a set interval.
- Aggregates data to provide a summary over the specified duration.

Process Tracking:
- Filters processes using pidstat based on a target command (e.g., next-server).
- Reports CPU and RAM usage per process, including average values.

Parallel Monitoring:
- Runs CPU and process monitoring concurrently to minimize delays.

Data Cleanup:
- Automatically removes temporary files after execution to keep the environment clean.

Requirements
- **SSH Connection:** Needed to place and activate the script on the virtual machine.
- **Sysstat:** Required to view CPU usage per vCPU, per core, and per application.

Installation
Connect to the virtual machine by running the command below in a terminal.

Replace [ip-address] with the IP address of the virtual machine [2].

```
ssh root@[ip-address]
```

Enter the password of the virtual machine.

Run the command below to install sysstat.

```
sudo apt install sysstat
```

K10 can be installed in two ways. The first is to connect to the Virtual Machine via SFTP, the second is to create the script via a text editor. The second way is explained in this appendix.

Run the command below to create the script:

```
sudo nano K10.sh
```

On your local machine, open K10 in a text editor.

Select all text and press CTRL + V.

Go to the Virtual Machine and paste the contents of the script into the new file.

Adjust the following properties if necessary:

---

[2] Deze is te vinden door in Vultr, onder Compute de test-node te selecteren.

```
INTERVAL=1                          # Hoe vaak er informatie wordt opgeslagen (sec)
DURATION=67                         # Hoelang er metrics moet worden verzameld
TARGET_COMMAND="next-server"        # Het process dat wordt gemonitord
```

It is recommended to do the test duration + ~7 seconds. This is because both scripts must be started manually, and it takes a few seconds before you have switched terminals.

To find out the process, you can use the command below:

```
pidstat
```

Save the file.

Execute the command below to make the script executable.

```
chmod +x K10.sh
```

Execute
Execute the command below to activate the script.

```
./K10.sh
```

Result
After executing the script, something like the tables below is shown.

The figure below shows the total vCPU usage and the memory usage in percentage per command.

```
# Process tracking (average stats):
# PID: PID      %CPU               Command      %Mem:
# PID: 5460     21.11              next-server  6.01
# PID: 14065    21.04              next-server  5.86
# PID: 19067    18.05              next-server  5.90
# PID: 22844    19.34              next-server  5.78
```

The figure below shows the usage per vCPU core.

```
# CPU %usr %nice %sys %iowait %irq %soft %steal %guest %gnice %idle
# all 80.22 0.00 6.41 0.00 0.00 4.56 0.00 0.00 0.00 8.81
# 0 80.22 0.00 6.41 0.00 0.00 4.56 0.00 0.00 0.00 8.81
```

In the case that the Virtual Machine has two vCPU cores, the figure looks like this.

```
# CPU %usr %nice %sys %iowait %irq %soft %steal %guest %gnice %idle
# all 80.22 0.00 6.41 0.00 0.00 4.56 0.00 0.00 0.00 8.81
# 0 40.11 0.00 3.20 0.00 0.00 2.28 0.00 0.00 0.00 4.41
# 1 40.11 0.00 3.21 0.00 0.00 2.28 0.00 0.00 0.00 4.40
```