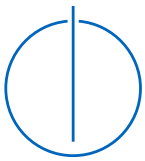


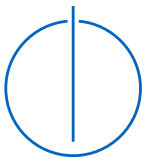
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

DevOps Versus NoOps Performance Comparison of an IoT Platform

Victor Pacyna



DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

DevOps Versus NoOps Performance Comparison of an IoT Platform

DevOps Versus NoOps Performancevergleich einer IoT-Plattform

Author:	Victor Pacyna
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	Mohak Chadha
Submission Date:	May 16th, 2022

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, May 16th, 2022 Victor Pacyna

Acknowledgments

I thank Prof. Dr. Michael Gerndt for the opportunity to write my thesis at his chair. I am grateful for the help from my advisor Mohak Chadah who assisted me with his patient support and invaluable feedback. On top of that, I want to thank my family and friends for their constant encouragement, unconditional sympathy and fruitful conversations.

Abstract

The emergence of large public cloud providers has led to a migration of corporate and consumer IT at revolutionary scale from private data centers to cloud infrastructure. Concurrently, software for orchestrating resources in the cloud as well as the application software had to adapt to the new environment. The microservice architecture quickly became the de facto standard of serverful cloud applications. Application development and operating cloud infrastructure were bundled into what is now commonly called DevOps. In recent years, the serverful cloud model is rivaled by the next stage of cloud computing – serverless computing. Application code can now be narrowed down even further into so called functions to abstract away any deployment details and focus purely on the application logic. The Function-as-a-Service (FaaS) model was introduced, DevOps turned into NoOps.

This thesis extends previous performance comparisons of DevOps and NoOps applications by transforming parts of a complex IoT platform, divided in several microservices, into a FaaS-based application and deploying them on the Google Cloud Platform as the testing ground. Performance tests are conducted utilizing cloud resources of production-level scale and simulated loads of real-world applications. In total four different deployment methods are investigated, two DevOps setups on Google's Kubernetes Engine with different auto-scaling configurations and two NoOps setups. The first NoOps setup uses Apache OpenWhisk (OW) – an open-source serverless platform – while the second uses Google Cloud Run (GCR) – Google's fully managed serverless compute platform. The results show the former performance advantage of serverful computing is reduced, if not equalized by more modern serverless offers like GCR. Simultaneously, a substantially cheaper pay-per-use cost model makes it a highly competitive cloud offering.

Contents

1. Introduction

Cloud computing is the corner stone of modern day corporate and consumer IT. For companies it provides a multitude of advantages – outsourcing of costly infrastructure, pay-per-use cost models, scalability and elasticity, fault tolerance and many more. Concurrent with the migration towards cloud infrastructure, new software architectures arose. Most prominently among them is the microservice architecture, that displaced the prevalent monolithic architecture. Its benefits closely align with those of the cloud. Newman [1] lists technology heterogeneity, robustness, scaling, ease of deployment, organizational alignment and composability as its core advantages.

But these innovations did not come without any downsides on their own and were quickly challenged by competing cloud offerings summarized under the term serverless computing. Development here is driven by public cloud providers like Amazon, Google and Microsoft with their respective products *AWS Lambda* [2], *Google Cloud Functions* [3] as well as *Google Cloud Run* [4] and *Azure Functions* [5]. They promise to simplify the developer experience without any concerns for the operational infrastructure (NoOps). With the serverless cloud model also the underlying programming paradigms change and microservice architectures were adapted to the new Function-as-a-Service (FaaS) model.

This bachelor thesis investigates these two competing software architectures tailored to applications deployed in the cloud and compares their performance in various aspects. As such it builds upon previous work from e.g., Fan, Jindal and Gerndt [6] or Jindal and Gerndt [7] among others. To conduct the experiment an IoT platform will be deployed in three different ways:

- Google Kubernetes Engine (GKE) with two different types of pod auto-scaling
- Apache OpenWhisk on top of GKE
- Google Cloud Run

The novelty of this study is based on the serverless technology used – Google Cloud Run – and different combinations of horizontal pod auto-scaling incorporated for the GKE deployment. The performance analyerveis is extended by an analysis that investigates the deployments' cost structures and their effect on per-request cost. It can therefore provide an approximation for production-level costs and aid decision making when specifying not only the the locus of deployment but also the software architecture.

2. Background

2.1. Microservices

Microservices are the manifestation and realization of an architectural style that has adapted to the needs of the cloud deployment of large applications. In contrast to a monolithic application, that is built as a single unit and can only scale horizontally – i.e. deploying the application in parallel on several severs behind a load balancer (see also paragraphs on GKE in subsection *Public Cloud Offering*) – an application, that is divided in several microservices, can be individually deployed and scaled – both horizontally and vertically. In short, "the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API" [8]. While there is no distinct definition of microservices, Lewis and Fowler [8] identified several characteristics exhibited by the majority of applications implementing the architecture:

- Componentization via services
- Organized around business capabilities
- Products not projects
- Smart endpoints and dumb pipes
- Decentralized governance
- Decentralized data management
- Infrastructure automation
- Design for failure
- Evolutionary design

Microservices reduce the complexity to operate the underlying infrastructure of previous monolithic applications. At the same time infrastructure and application logic are increasingly closer aligned to each other. These circumstances coined the notion of DevOps – short for Development and Operation – as a newly combined area of responsibility for the modern cloud engineer. Microservices are ideally deployed in isolation, a requirement which makes microservices heavily reliant on either *virtualization* or *containerization* as an enabling technology. Containerization is commonly preferred over virtualization as it is a more lightweight solution resulting in shorter spin up times and higher cost effectiveness [1]. In turn, wide-spread use of containerization has itself resulted in the need for orchestration tools like Kubernetes (K8s). As many other cloud computing tools, K8s was first developed internally – in that case by Google in 2014 – before it was open

sourced to the Cloud Native Computing Foundation (CNCF) in 2015. Orchestration is used to automate the deployment, scaling and management of containerized applications. Additionally, K8s offers a list of functions like [9]:

- Service discovery and load balancing
- Storage orchestration
- Automatic bin packing
- Self-healing
- Secret and configuration management

Kubernetes' architecture is rather minimalistic. The deployment of K8 always results in the creation of a cluster. Within a cluster worker machines, called nodes, run containerized applications. These workloads are hosted as pods on the nodes. Figure 2.1 visualizes the components of a cluster [10]. The *kube-apiserver (api)* exposes the Kubernetes API as its front end. *etcd* is a key-value store that contains all cluster configurations and data. The *kube-scheduler (sched)* selects the node on which a newly created pod will run. The *kube-controller-manager (c-m)* monitors all cluster resources like nodes, jobs, endpoints, services, etcetera and controls that the current state matches the desired state. Each node contains three components – the kubelet, the kube-proxy and the container runtime. The *kubelet* ensures that containers are running within pods and that the containers fulfill the pod's specifications. *kube-proxy (k-proxy)* is a network proxy and allows in- and outward communication for the pods. Finally the *container runtime* is responsible for running the containers on the node.

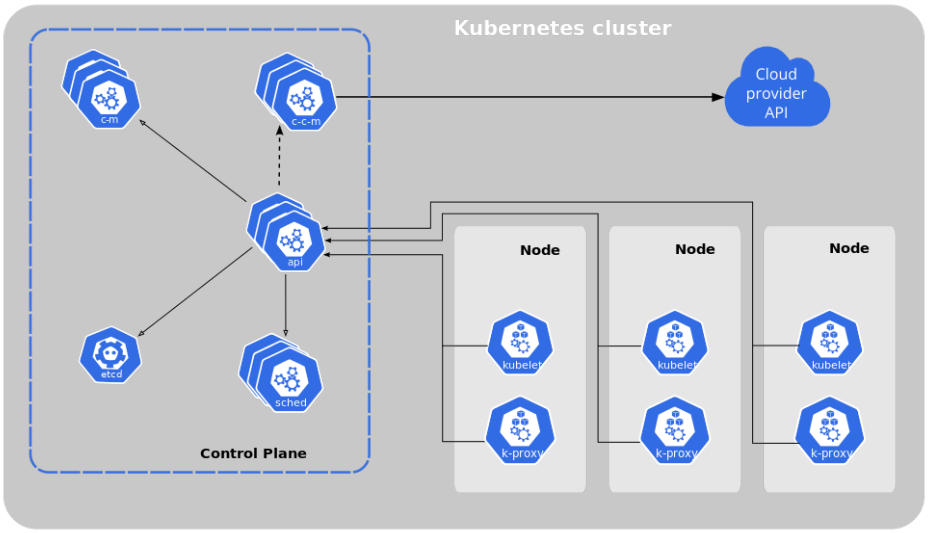


Figure 2.1.: Kubernetes Architecture [10]

2.2. Serverless Computing

Serverless computing is the second step in the revolutionary change brought to corporate and consumer IT, after the first migration from local data centers to public cloud providers, often coined with the notion serverful computing. While the first step targeted tasks of system administrators, serverless computing now targets the programmer of cloud applications. It is defined by three essential characteristics that are common to the various forms of serverless [11] :

1. An abstraction hiding the servers as well as the complexity to operate and program them (NoOps)
2. A pay-as-you-go cost model replacing the reservation-based model of serverful computing
3. Automatic, rapid and virtually unlimited up and down scaling of resources from zero to almost infinite.

The authors [11] divide serverless offerings broadly into Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS) (c.f. Figure 2.2). For this thesis the focus will lie upon the former. FaaS roughly dates back to the launch of AWS Lambda in November 2014. AWS Lambda's programming model and architecture are based on small code snippets – also called functions – that are easily deployed and executed in the cloud [2]. Most operational concerns such as resource provisioning, monitoring, maintenance, scalability, and fault-tolerance are left to the cloud provider. Nowadays every major cloud provider has its own FaaS offering. This range of function-based serverless offerings is extended by a few open-source projects that launched in the following years.

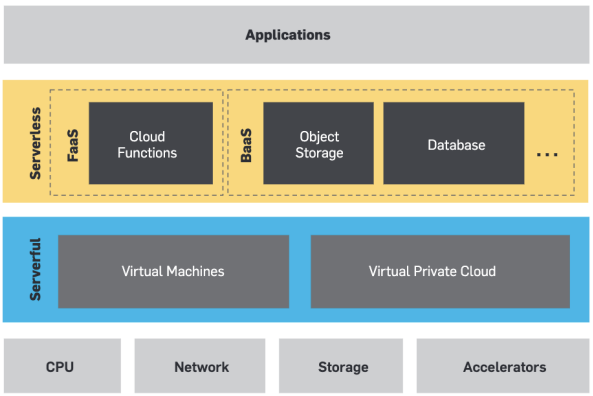


Figure 2.2.: Serverful vs. Serverless [11]

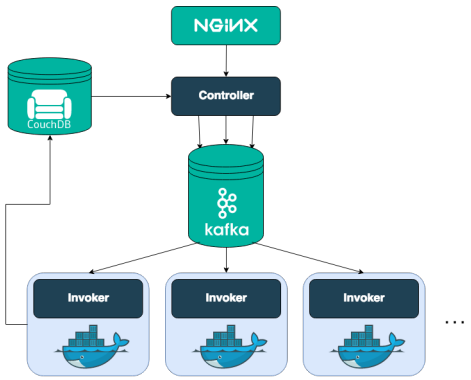


Figure 2.3.: OpenWhisk Architecture [12]

Apache OpenWhisk (OW) [12] is one of these open source distributed serverless platforms that executes functions (fx) in response to events at scale. Originally it was developed by IBM in 2015, released in 2016 and handed over to the Apache Software Foundation in 2019. The OpenWhisk platform supports a programming model in which developers write functional logic – also called actions – in any supported programming language, that can be dynamically scheduled and run in response to associated events – so called triggers – from external sources or from HTTP requests [12].

The IT architecture of OpenWhisk is depicted in Figure 2.3 [12]. OW's single point of entry is NGINX, an HTTP and reverse proxy server, whose main purpose is to forward incoming requests to the controller. The controller is the central component in the system's architecture. It matches the user's request to OW functionalities and checks authentication and authorization. In the case of an action invocation, it loads the action's code and configuration from CouchDB. A load balancer included in the controller checks for available executors, called invokers, and forwards the action invocation. The forwarding is done via Kafka (for further details see paragraph on Kafka in chapter 2.4) to prevent failures due to system crashes or high work loads. At this point, invocations are usually answered by an activation ID. This ID later on allows to extract the result from CouchDB once the asynchronous action execution in an isolated Docker container is completed. In case of web request though, the controller waits for the invocation's result.

2.3. Public Cloud Offerings

2.3.1. Google Kubernetes Engine

The Google Kubernetes Engine (GKE) is a cloud offering by Google that provides a managed environment for deploying, managing, and scaling containerized applications [13]. Multiple machines are grouped together to form a cluster, which are powered by the Kubernetes open-source cluster management system. Advantages are provided by Google Cloud's infrastructure like e.g., load balancing, node pools, auto-updates, auto-repair, and logging/monitoring capabilities. Additionally, it provides easy auto-scaling, which will be further explained in the next paragraph as it is of special interest for this work.

In general, there are two types of scaling in the context of cloud computing – horizontal and vertical scaling. Horizontal scaling describes the provisioning of more (or less) instances depending on the relevant performance metric like e.g. CPU or memory utilization. Vertical scaling refers to the increase (or decrease) in resources linked to an individual instance like vCPUs or GB of RAM granted to an instance. The GKE seizes upon this concept and provides four types of auto-scaling, which are illustrated in the 2x2 grid of Figure 1.2 [14]. The columns distinct between horizontal and vertical scaling,

while the rows separate pod- or node-based auto-scaling.

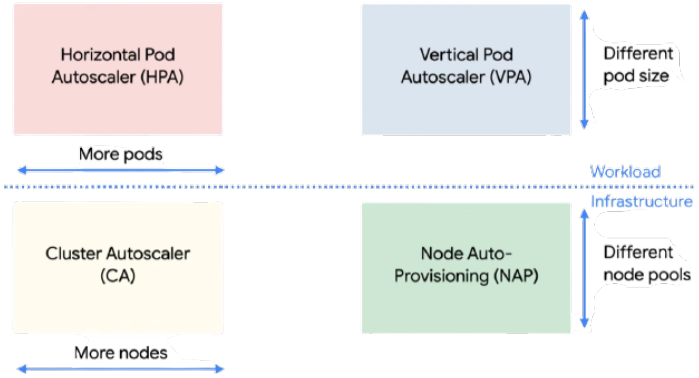


Figure 2.4.: Four Types of Auto-Scaling on GKE [14]

2.3.2. Google Cloud Run

Google Cloud Run [4] is a new serverless cloud offering by Google and was launched in 2019. It provides advantages to the complementary serverless offering Google Cloud Functions. It allows to run containerized functions on a fully managed compute platform, that are invocable through events or requests. The customer is separated from the infrastructure management and can therefore focus on application development. Google promises that any language, any library and any binary can be run and thereby tries to ensure portability to prevent vendor lock-ins, a feature that many other serverless offering cannot provide.

Cloud Run is closely linked to Google's engagement in the KNative open source project [15] as it implements the KNative API. KNative is supported by various industry leaders like Google, Red Hat and IBM with the goal to deploy and run serverless applications on any Kubernetes platform. It consists of three primary components – Build, Serving and Eventing. As it can be allocated on any Kubernetes cluster, it provides scalable, secure, stateless services without the vendor lock-in [16].

2.4. IoT Platform

The application that serves as the real-life testing ground for the performance comparison of microservice and serverless deployments in this thesis is an IoT platform developed at the Chair of Computer Architecture and Parallel Systems at the Technical University of

Munich (TUM). Its purpose is to provide an IoT infrastructure for device, sensor or user administration and persistent storage for sensor data that is received via HTTP, WebSocket or MQTT protocol. This main functionality is extended by secure communication, scalability, platform independence and analytical extensibility. The following sub-section will explain the data model and the microservice architecture of the IoT platform, before a selection of endpoints will be introduced, that were subject of this thesis‘ load testing.

2.4.1. Data Model & IT Architecture

The IoT platform's data model contains four entities of interest for this thesis – users, devices, sensors and consumers. Users are the interacting entity with the platform. Their attributes are `name`, credentials consisting of a `username` and a `password` and a `role`, which is either admin or user. A super user is globally defined and reserved for the highest administrative purposes. Each user can have associated multiple devices linked to his or her account, which in turn then can have multiple sensors linked to each device. Attributes of those can be seen in the following Figure 2.5. Consumers are the platform's mode to retrieve sensor data. A user can have multiple consumers, that each allow access to a self-administered set of the user's sensors. Prior to retrieving data, the user has to grant a consumer access to one or multiple sensors.

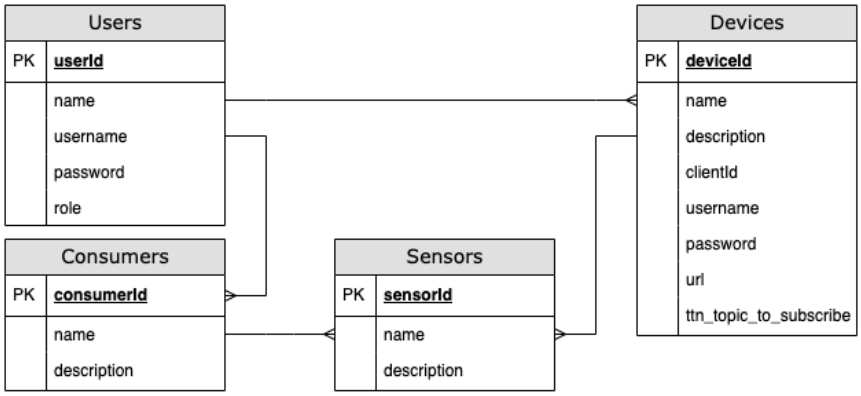


Figure 2.5.: Data Model of the IoT Platform

The platform has several key components, that combined as a service mesh (c.f. Figure 2.6) provide the formerly described functionality. In its center lies the IoT core that serves both the front- and backend to all managerial tasks. Users, devices, sensors and more can be managed from here. It also provides authentication tokens for secure communication

from the outside. Persistent data of user, device or sensor instances is stored on MariaDB, an "open-source, multi-threaded, relational database management system (DBMS)" [17] that originated from a fork of the open-source DBMS MySQL.

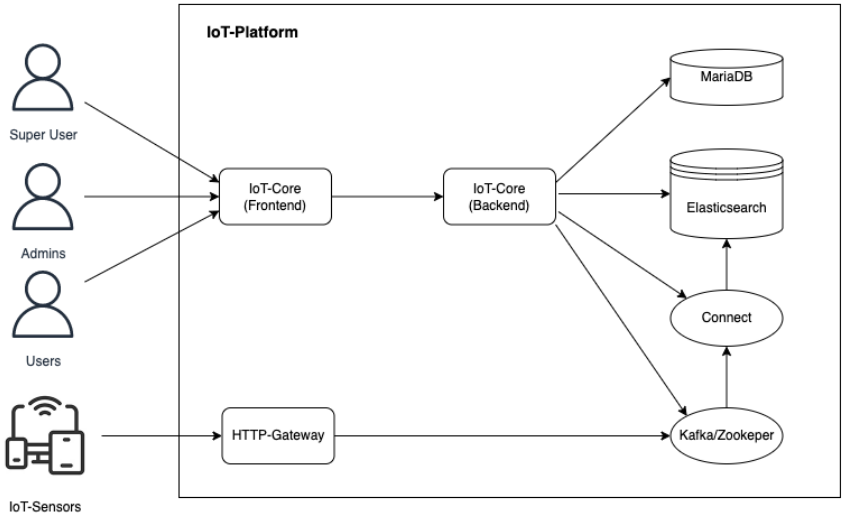


Figure 2.6.: Architecture of the IoT Platform

Sensor data enters the system through gateways. Originally three alternatives – HTTP, WebSocket and MQTT – were developed by the chair , while this thesis will only utilize the HTTP gateway due to its focus on serverless computing which contradicts MQTT's and WebSocket's statefulness and component longevity. The gateway's main purpose is to check the authentication of the sensor and verify the data schema. After a successful check the received data is the forwarded to Kafka.

Kafka is a distributed streaming platform, released under the Apache software license, that allows to store and process data streams at scale [18]. It incorporates a publish-subscribe pattern and holds the role of a broker in that design. One or multiple producers publish data records to a topic, the core abstraction of Kafka that is used to store and process streams of records. A single topic then can be subscribed by zero, one or multiple consumers (subscribers).

The IoT platform utilizes Kafka by matching each sensor to a topic. Sensor data is then logged under its topic, before it is further passed on via a connector to Elasticsearch. As such it represents a fault-tolerant and – for a short period of time – durable message queue within the architecture of the IoT-platform.

At the end sensor data is persistently stored in Elasticsearch. It is an open-source

distributed search and analytics engine built upon Apache Lucene utilizing its indexing and search features [19]. It is commonly used for logging and log analytics, infrastructure metrics and container monitoring. Its advantages of speed, scalability and resiliency through distribution are the main reasons for its usage on the platform. It can ingest raw data from various sources, in case of the IoT-platform from Kafka. Data then is parsed, normalized and enriched before being indexed. An Elasticseach index stores JSON documents in a data structure called inverted index, that allows near real-time searches. In the application of the IoT platform one index is created for each sensor. Sensor data is then retrieved from the IoT core by consumer instances that query the index.

2.4.2. Endpoints

`Users-Signin` **Endpoint** */api/users/signin*

This endpoint is central for the authentication and authorization on the IoT platform. A users sends a POST request containing his or her username and password as parameters. Internally these credentials are compared to the values stored on MariaDB. Should this check be successful, a JSON web token (JWT) is returned. JWT is "a compact, URL-safe means of representing claims to be transferred between two parties" [20]. It is useful for the IoT platform as it is stateless by design and therefore does not have to be stored on the server. It securely encodes the user's ID. With every request the user passes it as part of the authorization header to the server, where it can then be quickly verified.

`Users-Get` **Endpoint** */api/users/*

The `Users-Get` endpoint is only requested with the JWT in the authorization header. No additional parameters are required for the GET request. The IoT core checks the authentication and then, if successful, returns the user details from MariaDB. Should the user be logged in as the super user, a list of all users is returned.

`Devices-Add` **Endpoint** */api/users/* : *user_id/devices*

Besides the JWT authorization, this endpoint requires an user ID as a request parameter. Additional parameters are the values for the new device's attributes. As such only the device name is required, while the others are optional (c.f. Figure 2.5). The IoT core checks the user and then inserts the new device into the DB. It returns the successfully created device in the response body.

`Devices-Get` **Endpoint** */api/users/* : *user_id/devices*

Similar to the `Users-Get` endpoint, the `Devices-Get` endpoint returns all devices associated with the user ID provided as the required request parameter. Again, it verifies the JWT, checks the user and finally returns all devices associated with the user from MariaDB.

`Sensors-Add` **Endpoint** */api/users/* : *user_id/devices/* : *device_id/sensors*

In part the functionality of the `Sensors-Add` endpoint is congruent to the `Users-Add` and `Devices-Add` endpoint respectively. The POST request is sent with the JWT token, a user ID and a device ID. Additionally, the sensor attributes, name and description, are passed, of which the former is required. After the sensor is inserted into the MariaDB, three further processes are initiated by the IoT core. Recapitulating the architecture of the platform, one sensor is matched by a Kafka topic, an Elasticsearch index and the connect job between those two. The IoT core creates all of those, while simultaneously returning the newly created sensor back in the response body.

`Sensors-Get` **Endpoint** */api/users/* : *user_id/devices/* : *device_id/sensors*

The `Sensors-Get` endpoint behaves very similar to the early introduced endpoints. The GET request, containing a JWT in the header, a user and device ID, is answered by querying the database for matching sensors and returning them as a list in the response.

`HTTP Gateway`

The HTTP gateway's purpose is different to those of the IoT core endpoints. While later mainly serve an administrative aim, the gateway is essential for the platform's capability to receive sensor data. Sensor data is sent from an IoT device in a POST request's body, accompanied by a JWT device key in the authorization header. Sensor data can be one or multiple messages, that contain the sensor ID, a value and a timestamp. Once verified, the gateway creates a Kafka producer and sends the messages to the dedicated topic. A successful transmission of all messages is signaled with a 200 response code to the IoT device.

`Consumers-Consume-Get` **Endpoint** */api/consumers/consume/* : *sensor_id*

The `Consumers-Consume-Get` endpoint is the IoT core's principal way of retrieving sensor data. Consumers, similar to devices, have JWT keys. A request to this endpoint has to contain the consumer key in the authorization header as well as the sensor ID as a parameter. These two are verified and, if successful, a query is sent to Elasticsearch for retrieving documents (i.e. sensor logs) of the respective index. The result of that query is then returned in the response body.

3. From Microservice to Function-as-a-Service

With the rise of serverless computing, also academia's interest in its applications, advantages and disadvantages has risen. Studies that have been previously conducted often utilized smaller applications like the movies service [7]. Jin et al. [21] criticize this approach and argue that the demand for applying serverless computing to more complex and stateful applications is increasing. The restructuring of existing applications is mostly time consuming and ad-hoc. The authors [21] identified two main objectives of such a migration – it must be cost effective, while maintaining comparable performance at the same time. These goals result in a need for maximum code re-usage and a principal approach for migrations from microservices to serverless functions.

This bachelor thesis aims to compare more complex microservice and serverless deployments as the previous introduction to the IoT platform's architecture has shown. The following chapter will give an exemplary overview on how single endpoints of the IoT core were migrated to both OpenWhisk and Cloud Run. It will highlight advantages and disadvantages of modifications for both platforms and tries to summarize a few principles that could be observed progressing from microservice to serverless computing.

The starting point of the transition from microservice to serverless function were the Node.js [22] applications for the IoT core and the HTTP gateway microservice. The following chapter will only showcase the transformation of one specific endpoint, namely the `Sensors-Get` endpoint, while the other endpoints were adapted accordingly. Within the IoT core the web-application framework Express [23] is used to serve incoming requests. A router checks on which endpoint a request was received and then forwards it to the designated middle-ware functions. In case of the `Sensors-Get` endpoint it is forwarded firstly to the authentication handler and then to the sensors controller.

For the OW deployment these two middle-ware components were extracted into individual functions. Starting with an existing Node.js-action Docker image, custom action runtimes were created. They contain only the required Node.js packages required for the authentication and the sensor controller action respectively. Additionally the custom Docker action runtimes contain key files, connection configurations and SQL schema whereever necessary. Important to note here is that it reduces cold start latencies by

including only what is absolutely necessary. While the Docker image of the IoT core contains every package the Express server needs to answer requests on all endpoints, Docker images for serverless functions should be reduced to a minimum.

Application code could be reused to a large extent as the controller class' own functions mirrored closely what was required by the single OW functions. An advantage of the OW platform was its easy-to-use implementation of action sequencing. By creating two separate actions for the authentication and the `Sensors-Get` functionality and chaining them together, the authentication function could be reused across multiple action-sequences [24]. The resulting Docker images were therefore a bit smaller in size. In a real-life, production-level deployment it would also become more likely to encounter warm authentication action invocations, i.e. a running instance is already present on the the FaaS platform, as it is part of multiple sequences instead of being bound to only one individual action.

The GCR transformation proceeded in a similar fashion. Code re-usage was even higher as it utilizes the Express framework resembling the IoT core microservice. Keeping the Express framework, Google's instructions on how to provide a custom Node.js service were followed [25]. Unfortunately function sequencing is offered as its own cloud service, called Google Cloud Workflows [26], which is billed separately and therefore deemed out of scope for this bachelor thesis. Instead authentication and endpoint logic were sequenced within the functions code.

Deployments were completely realized with the help of the Docker images. While the Docker images for the OW actions had to be hosted on a publicly available DockerHub repository – this can be considered a security risk if used carelessly by e.g. storing keys or pulic URLs in the image – the Cloud Run deployment was conducted using Google's Cloud Build service [27] and artifact registry [28]. Action-level configurations like concurrency limits and memory usage were defined through the command-line interface of OW and GCR respectively.

4. Methodology

4.1. Research Questions

The common notion found in research on serverless computing is that functions, that are either compute-intensive or serve highly fluctuating demand, are well suited for deployment on serverless platforms (see e.g. [7]; [6]). This bachelor thesis extends these evaluations by an in-depth performance analysis of a real-world IoT platform. It investigates critical paths in the application and locates strengths and weaknesses of both microservice and serverless deployments. Research questions that this thesis addresses are:

RQ1: Which tasks on an IoT platform are better suited for microservices, which for FaaS deployments?

RQ2: What do request patterns look like that utilize the capabilities of either microservice or FaaS platforms?

RQ3: How do Google's auto-scaling mechanisms effect the performance of the IoT platform in the microservice deployment?

RQ4: How does Google's container-based serverless cloud offering Cloud Run compare to the self-deployed open-source alternative OpenWhisk?

RQ5: How do the platforms differ in terms of their pricing models and the resulting total cost as well as relative cost per request?

4.2. Deployment Strategies

Overall, three different deployment strategies – with variations in their auto-scaling capacities – were tested. These are:

- Google Kubernetes Engine (GKE)
- Apache OpenWhisk (OW) on top of GKE
- Google Cloud Run (GCR)

In a first step it had to be guaranteed that the serverless deployments and the microservice deployment remained comparable and that fairness in terms of resource provision was

ensured. All services that were not transformed into functions were deployed to a Kubernetes cluster with a fixed resource configuration of three VMs with 4 vCPUs, 16 GB of RAM each. This cluster contained all components except the IoT core and the gateway (c.f. Figure 2.6). All services were exposed using internal load balancers that make them accessible only within the project's virtual private cloud (VPC). This also guaranteed that access times were comparable between all three deployments.

In a second step the services under investigation had to be deployed both as DevOps and NoOps deployments. The microservice setup was created using Google's Kubernetes Engine in standard mode. In comparison to GKE's autopilot mode it allows to specify the configuration of the worker nodes. The GKE cluster was setup with a node pool of two virtual machines of 8 vCPUs, 32 GB of RAM and 100 GB persistent disk memory, that would auto-scale to five nodes if necessary [29]. The IoT core and HTTP gateway deployments were configured to request half a vCPU and use a maximum of one vCPU per pod. A horizontal pod autoscaler (HPA) [30] allowed the pods to scale up to 40 pods, according to two different scaling strategies. In one configuration the scaling metric's target was set to 50% CPU utilization and in the second one to 80% CPU utilization. These two setups are called GKE-50 and GKE-80 in the following work.

The OpenWhisk setup was deployed on top of a GKE cluster as well using Helm, an open-source Kubernetes package manager hosted by the CNCF [31]. It is the recommend deployment method for larger OW setups. OW and GKE node autoscaler unfortunately did not work together, so that no additional nodes were created under high resource utilization. To keep the experiment results comparable, it was decided to set the number of nodes to the maximum of five nodes from the start of each experiment run. The reason for this incompatibility most likely lies in the lack of predefined resource requests of the OpenWhisk components. GKE's cluster autoscaler needs these to scale the number of nodes automatically. The configurations of OW's components were furthermore adjusted to serve high loads of the performance tests following deployment hints from the documentation [32]. The controller and invoker count was set to two, concurrency was allowed, container pool user memory was set to 10 GB, heap space was increased to 1 GB for the invoker and to 2 GB for the controller, logging was turned off, action invocations were limited to 25,000 per minute and maximum concurrent invocations were set to 9,999.

The Google Cloud Run setup was configured to use one vCPU per instance with a maximum 20 instances per service. The concurrency limit was set to 100 and the memory configuration was 256 MB. The Cloud Run functions were connected to the shared services with a serverless VPC connector [33], that allowed access to the internal load balancers of the shared services.

4.3. Test Design

Load testing was performed with the help of the K6 web-app testing tool [34]. The documentation describes it as "a developer-centric, free and open-source load testing tool built for making performance testing a productive and enjoyable experience" [34]. It offers a variety of load testing capabilities. Tests in K6 are based on virtual users (VUs) that try to perform a given test as often as possible, comparable to a WHILE-loop. One can configure the number of VUs, the way they are added (linear, instantaneous, etcetera) over time and the duration of a testing stage. Chaining a few of these stages together, so-called scenarios can be created. This study utilized three different scenarios to mimic different workloads:

- Linear: A 30min linear increase to a target of 100 VUs.
- Random: 60 VUs at 7min, 30 VUs at 14min, 100 VUs at 21min, 40 VUs at 28min and 0 VUs after 30min. Transitions between targeted values are linear.
- Spike: After 1min a plateau of 10 VUs is reached. Between 14min and 16min a spike of 100 VUs is created. Following the spike, the plateau of 10 VUs is held for another 13min before the VUs are linearly decreased to 0.

To prepare the experiment runs on the different endpoints a set of 100 users with five devices and one sensor each were created on the IoT platform. For each test request a random instance was picked from a previously extracted list. Caching should not bias the results as the Node.js packages used for connecting to MariaDB and Elasticsearch do not implement caching behavior as their default.

For the execution of the tests a VM was created on TUM's self-hosted cloud, the LRZ-Cloud. On this VM the single tests were scheduled in a queue with 15-30 min time buffers in-between to allow the clusters to scale back to their initial pod, instance and node counts. The results were stored in two different ways. Results with the granularity of of individual requests were stored as CSV files, while the overall test metrics generated by K6 were stored as standard text files.

5. Results

This chapter reports the outcome of all 22 experiment runs on eight endpoints. On seven of them (`Users-Signin`, `Users-Get`, `Devices-Add`, `Devices-Get`, `Sensors-Get`, `HTTP-Gateway` and `Consumer-Consume-Get`) the three workload patterns described before were executed. A special case was the `Sensors-Add` endpoint. Here a scaling issue occurred that had to be met with a change in the testing procedure. The IoT platform was not able to handle several hundreds of sensors being added. This was not an issue caused by program code of the platform itself but the backend services of Elasticsearch and Kafka. The provided modes of deployment did not scale to multiple nodes which is necessary when larger amounts of indices and topics respectively are created. This is caused by complex data structures and redundant copies required in the background. It was deemed out of scope for this bachelor thesis to adapt the backend services to handle high workloads. Instead it was decided to only conduct a five minute experiment run on the critical endpoint, which will later on be called sprint workload.

The chapter is divided into two sub-chapters, the performance and the cost analysis. The first sub-chapter will display the performance metrics of each experiment run. These are an overview of request counts containing the total amount of requests answered, the number of failed requests and the resulting goodput. Goodput is a metric that was introduced to report the percentage of responses that were answered with a status code of 200 for a successful request. The second group of metrics are response time based, where the average, minimum, median, maximum and the 90%-percentile as well as the 95%-percentile are reported. These metrics, aggregated over the whole length of an experiment run, will be extended by three visualizations on how request count, p(95) and goodput evolved over the course of a 30 min experiment. For these visualizations the values of the metrics were sampled on 10s intervals.

The second sub-chapter will report the costs that occurred conducting the experiments. It will further explain how these costs were calculated for each deployment method and which method was favourable under given conditions.

5.1. Performance Analysis

5.1.1. `Users-Signin` Endpoint

For the linear workload on the `Users-Signin` endpoint the fastest deployment was OW with an average response time of 351.22ms, followed by GCR with 380.58ms, GKE-50 with 479.57ms and GKE-80 with 518.59ms. The number of requests served differed quite noticeably as well. While OW could serve 256,060 responses in 30min, GCR served 236,034 and the serverful deployments served 187,615 requests in the case of GKE-50 and 173,522 in the case of GKE-80. For GKE-50 there were two occasions as Figure 5.1, in which goodput fell by more then 10% and also the requests served dropped, while p(95)-response time rapidly increased. This could have been either caused by a networking issue or something failed during auto-scaling as request timeouts of 1min occurred. Interestingly OW shows substantially less high-frequent noise in its p(95) response time graph than the other deployments. It outperforms the others which is contradicting to previous studies. One reason for it could lie in its over-provisioning of five nodes from the start of the experiment run. GKE-50 and GKE-80 only scaled to three nodes and did not utilize the full resources they were entitled to.

Table 5.1.: Overall Request Counts of Linear Workload on `Users-Signin` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	62	187,615	99.967%
GKE-80	0	173,522	100%
OW	0	256,060	100%
GCR	2	236,034	99.999%

Table 5.2.: Overall Request Durations of Linear Workload on `Users-Signin` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	479.57ms	110.69ms	385.44ms	1m0s	841.36ms	1.05s
GKE-80	518.59ms	111.17ms	343.3ms	3.11s	1.16s	1.49s
OW	351.22ms	125.8ms	318.88ms	9.5s	566.87ms	650.64ms
GCR	380.58ms	17.98ms	184.37ms	4.73s	922.96ms	1.42s

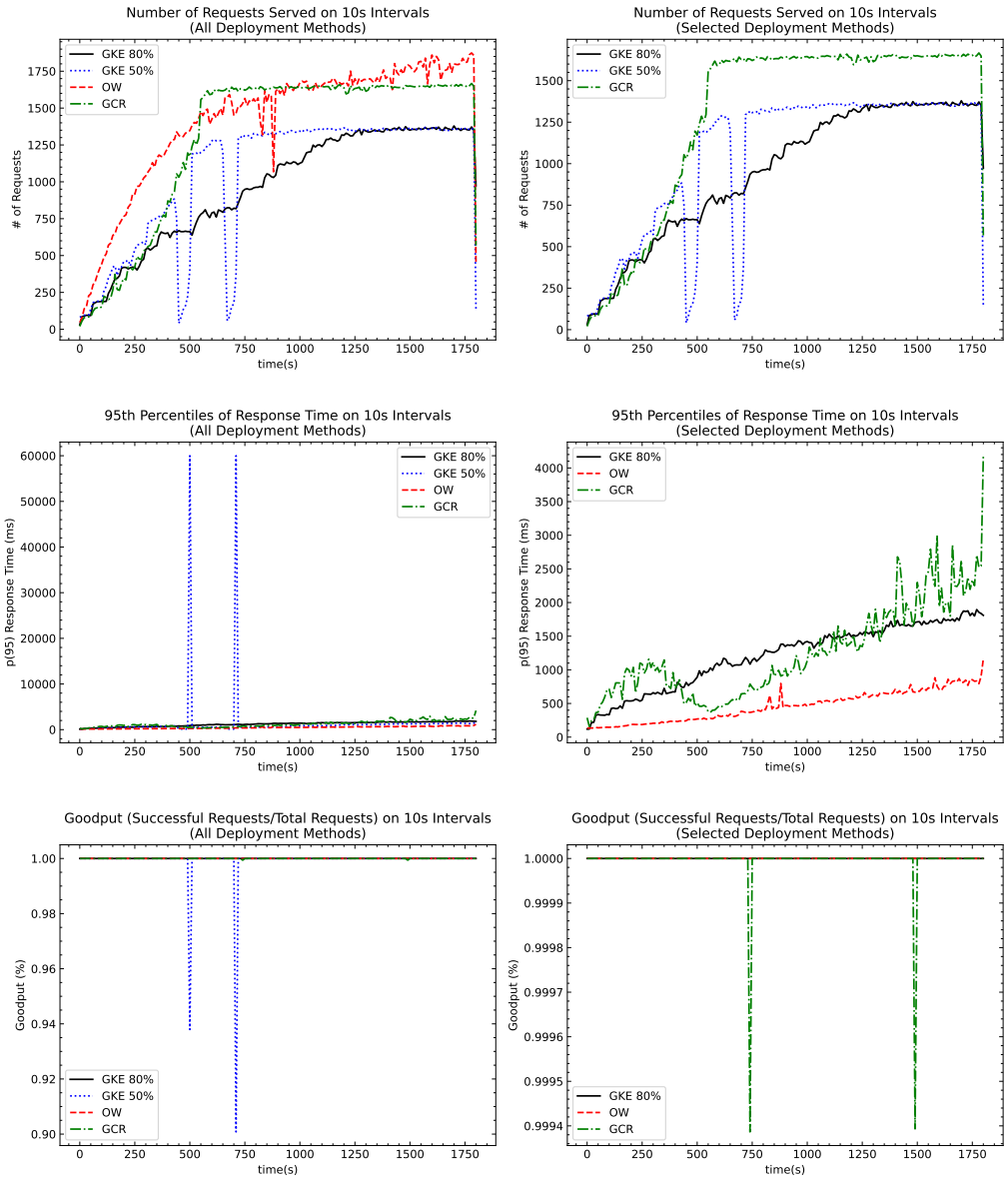


Figure 5.1.: Results of Linear Workload on `Users-Signin` Endpoint

The results for the random workload were tainted by a serve failure of GCR. The overall goodput fell to only 60.733%. The high amount of quickly failed requests of 146,168 also inflated amount of requests served totally to 372,242. The average response time dropped to 243.34ms. The logs did not reveal the cause of this failure. The assumption is that something in GCR auto-scaling mechanisms failed as the event coincides with the first workload peak.

For the other deployments the results matched those from the linear workload. GKE-50 served 189,695 requests within 30min with an average response time of 478.93ms, GKE-80 155,997 requests with an average response time of 582.41ms and OW served 259,701 requests with 349.62ms on average.

Table 5.3.: Overall Request Counts of Random Workload on `Users-Signin` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	0	189,695	100%
GKE-80	0	155,997	100%
OW	0	259,701	100%
GCR	146,168	372,242	60.733%

Table 5.4.: Overall Request Durations of Random Workload on `Users-Signin` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	478.93ms	109.72ms	347.34ms	3.6s	980.62ms	1.32s
GKE-80	582.41ms	107.6ms	377.23ms	5.63s	1.1s	2.25s
OW	349.62ms	125.53ms	320.17ms	5.36s	540.56ms	619.92ms
GCR	243.34ms	0.01ms	139.72ms	10.14s	397.27ms	999.18ms

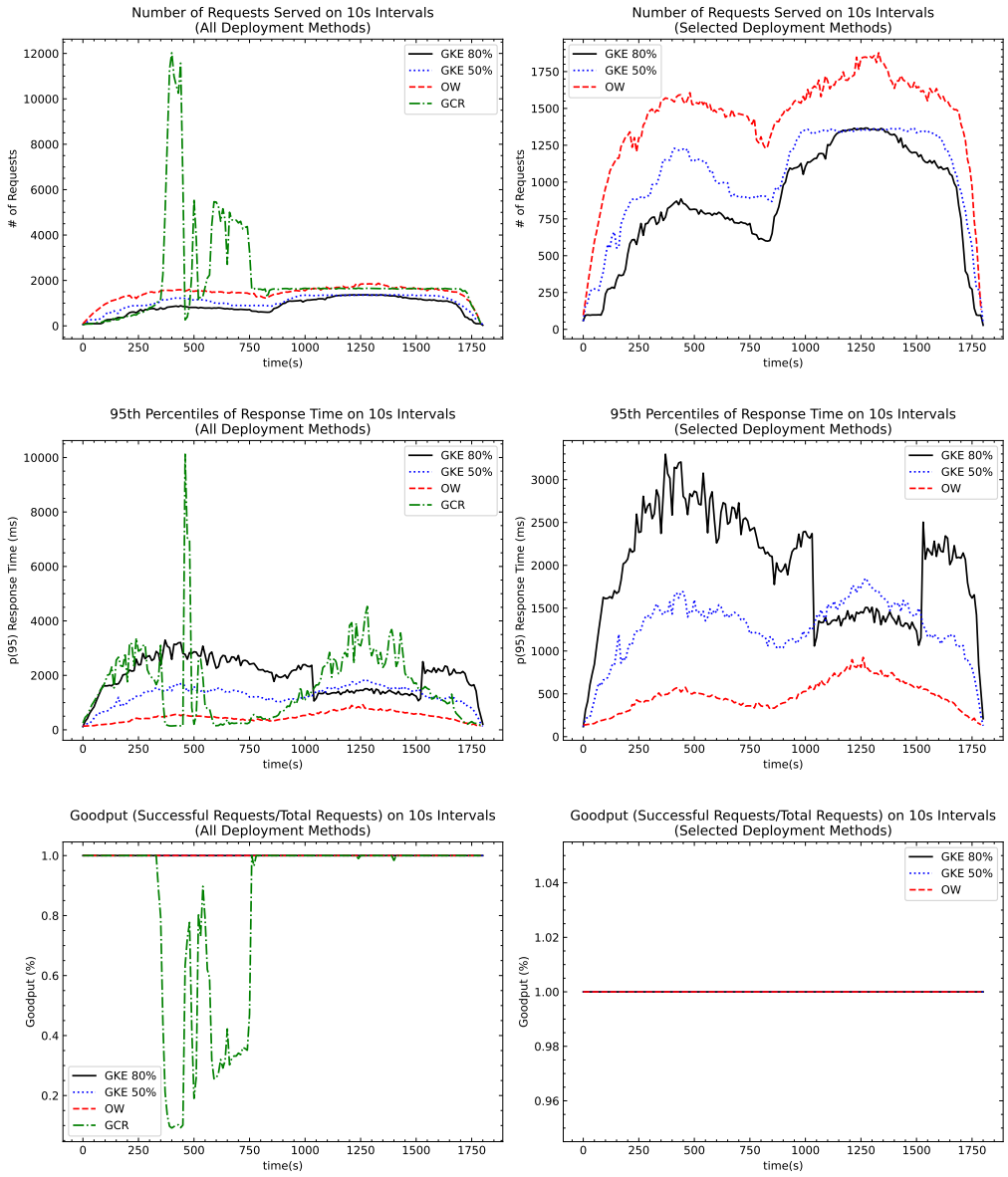


Figure 5.2.: Results of Random Workload on `Users-Signin` Endpoint

The spike workload surprisingly showed a large difference between the two GKE setups. While GKE-50 served 53,191 requests within 429.77ms on average, GKE-80 served only 18,974 requests with an average response time of 1.2s. Figure 5.3 shows that this difference were not only visible during the spike in workload, but also during the constant workload.

OW was again the fastest deployment method with 124,531 requests answered within 177.37ms on average. GCR served 88,638 requests of which 152 failed with an average response time of 257.14ms.

Table 5.5.: Overall Request Counts of Spike Workload on `Users-Signin` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	0	53,181	100%
GKE-80	0	18,974	100%
OW	0	128,531	100%
GCR	152	88,638	99.829%

Table 5.6.: Overall Request Durations of Spike Workload on `Users-Signin` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	429.77ms	108.58ms	427.49ms	2.98s	596.25ms	914.38ms
GKE-80	1.2s	112.78ms	1.07s	5.48s	1.72s	2.68s
OW	177.37ms	124.64ms	142.39ms	6.18s	253.23ms	402ms
GCR	257.14ms	24.47ms	158.33ms	8.66s	351.5ms	469.9ms

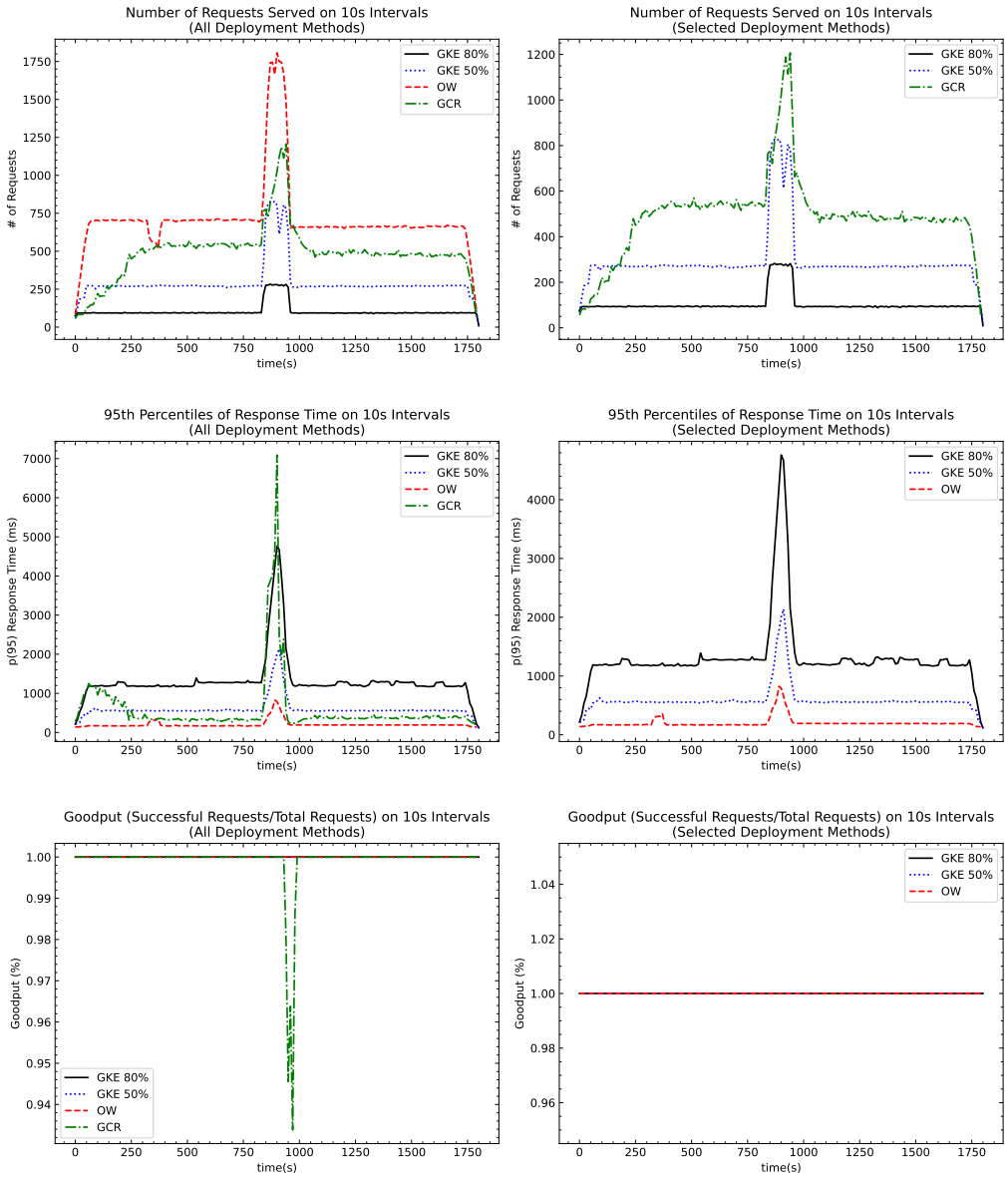


Figure 5.3.: Results of Spike Workload on `Users-Signin` Endpoint

5.1.2. `Users-Get` Endpoint

On the `Users-Get` endpoint the GKE deployments rivaled and even exceeded the performance of the serverless deployments. GKE-50 served 327,921 requests within 23.7ms on average, GKE-80 served 324,931 requests within 26.2ms. OW could not keep up with only 154,893 requests served with an average response time of 332.29ms, while GCR could match GKE performance by answering 325,235 requests with an average response time of 25.94ms.

OW seemed to reach a performance limit at around 2,000 requests per 10s, where response time increased sharply. Figure 5.4 (middle row, right column) shows well how GCR's cold start latency drops within 10s.

Table 5.7.: Overall Request Counts of Linear Workload on `Users-Get` Endpoint

Deployment Method	Failed	Total	Goodput
GKE-50	0	327,921	100%
GKE-80	0	324,931	100%
OW	9	154,893	99,994%
GCR	0	325,239	100%

Table 5.8.: Overall Request Durations of Linear Workload on `Users-Get` Endpoint

Deployment Method	AVG	MIN	MED	MAX	p(90)	p(95)
GKE-50	23.7ms	12.49ms	23.41ms	123.67ms	32.93ms	36.5ms
GKE-80	26.2ms	12.42ms	24.35ms	237.45ms	39.52ms	47.9ms
OW	332.29ms	26.29ms	40.63ms	30.88s	85.52ms	145.88ms
GCR	25.94ms	16.73ms	24.04ms	1.86s	34.23ms	38.36ms

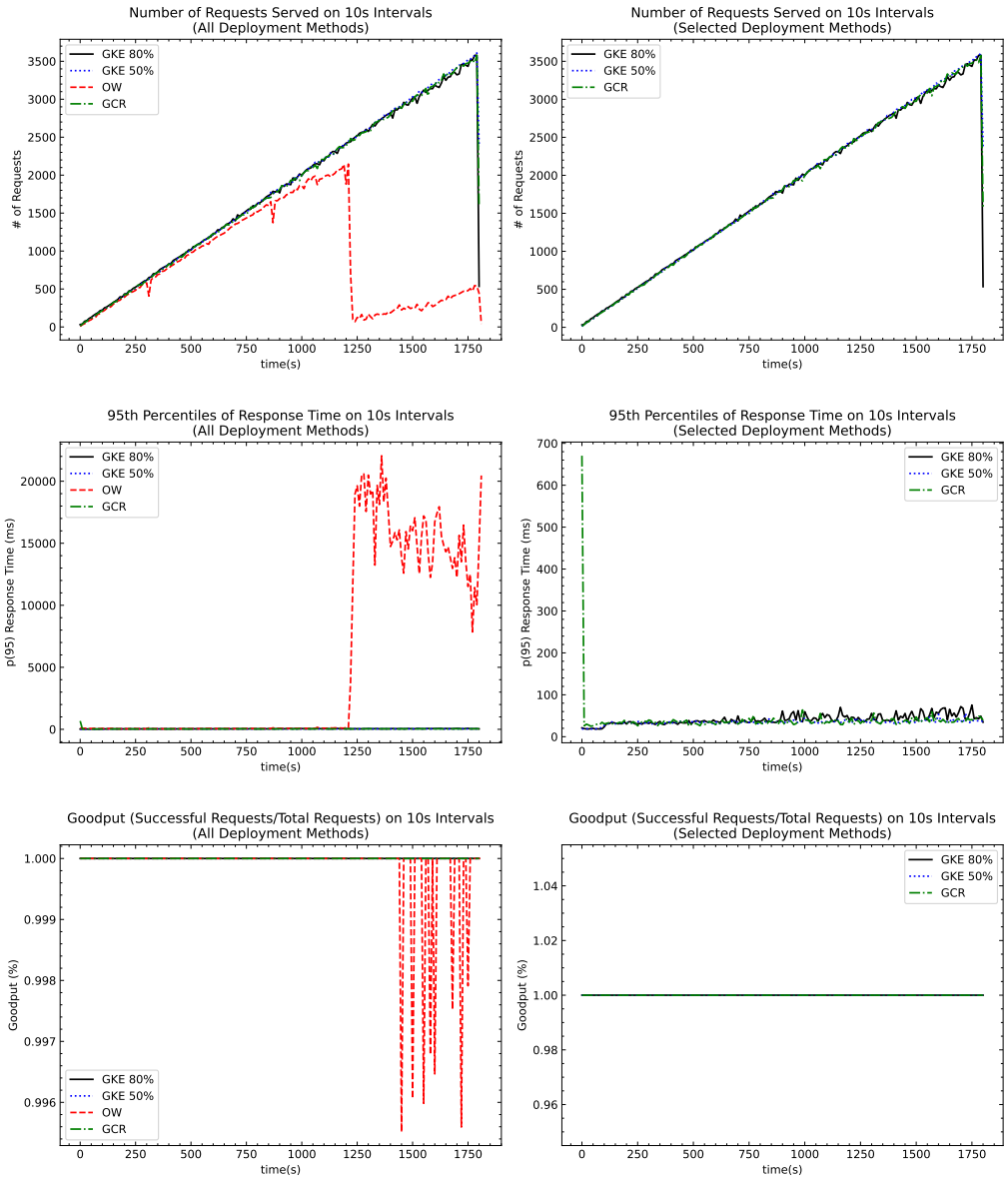


Figure 5.4.: Results of Linear Workload on `Users-Get` Endpoint

The random workload showed a similar pattern of results as the linear workload. While GKE-50's (329,417 requests, 25.09ms), GKE-80's (328,549 requests, 25.82ms) and GCR's performance (330,382 requests, 24.28ms) are nearly on par, OW falls behind with 259,701 requests and an average response time of 590.08ms. The performance limit seem to be at the same threshold as before, roughly at 2,000 requests per 10s.

Table 5.9.: Overall Request Counts of Random Workload on `Users-Get` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	1	329,417	100%
GKE-80	0	328,549	100%
OW	0	259,701	100%
GCR	0	330,382	100%

Table 5.10.: Overall Request Durations of Random Workload on `Users-Get` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	25.09ms	0s	23.69ms	276.44ms	36.29ms	42.25ms
GKE-80	25.82ms	12.57ms	24.54ms	250.19ms	37.84ms	44.07ms
OW	590.08ms	26.12ms	35.3ms	1m0s	98.09ms	2.29s
GCR	24.28ms	15.9ms	22.75ms	233.3ms	31.35ms	35.58ms

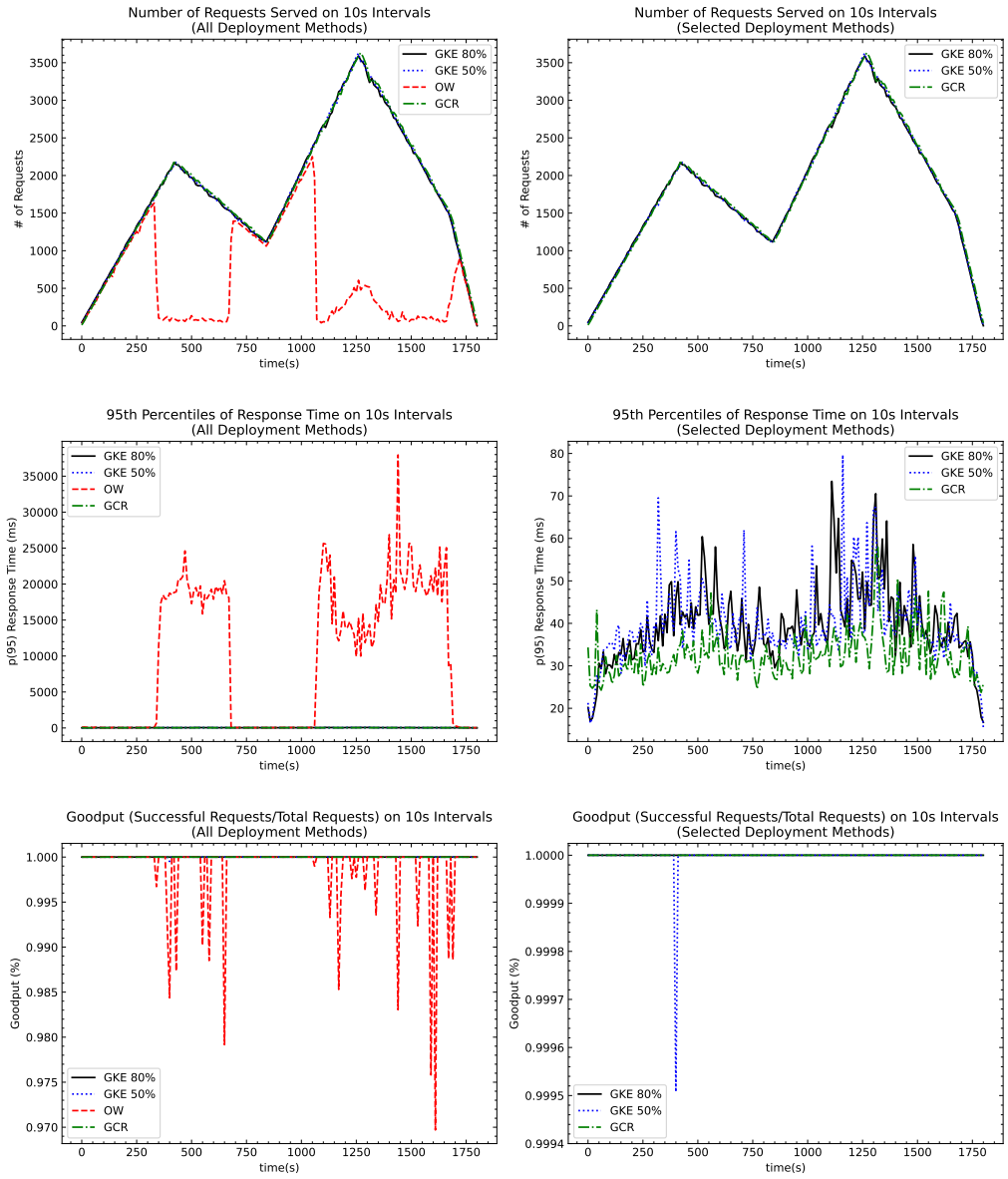


Figure 5.5.: Results of Random Workload on `Users-Get` Endpoint

For the spike workload the patter stayed the same. GCR leads with 83,120 requests served within 23.94 ms on average, followed by GKE-80 with 82,791 requests and 25.09ms and GKE-50 with 81,725 requests and 28.69ms. OW could only serve 61,472 requests and it took on average 127.9ms to respond. Figure 5.6 shows that GCR scaled so quickly that p(95) response times barely spiked during the sharp workload increase.

Table 5.11.: Overall Request Counts of Spike Workload on `Users-Get` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	1	81,725	99.999%
GKE-80	0	82,791	100%
OW	1	61,472	99.998%
GCR	0	83,120	100%

Table 5.12.: Overall Request Durations of Spike Workload on `Users-Get` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	28.69ms	0s	26.07ms	231.06ms	39.72ms	62.5ms
GKE-80	25.09ms	12.58ms	23.42ms	244.72ms	33.55ms	43.56ms
OW	127.9ms	27.12ms	33.85ms	34.23s	40.86ms	52.44ms
GCR	23.94ms	16.33ms	23.16ms	1.97s	28.65ms	32.75ms

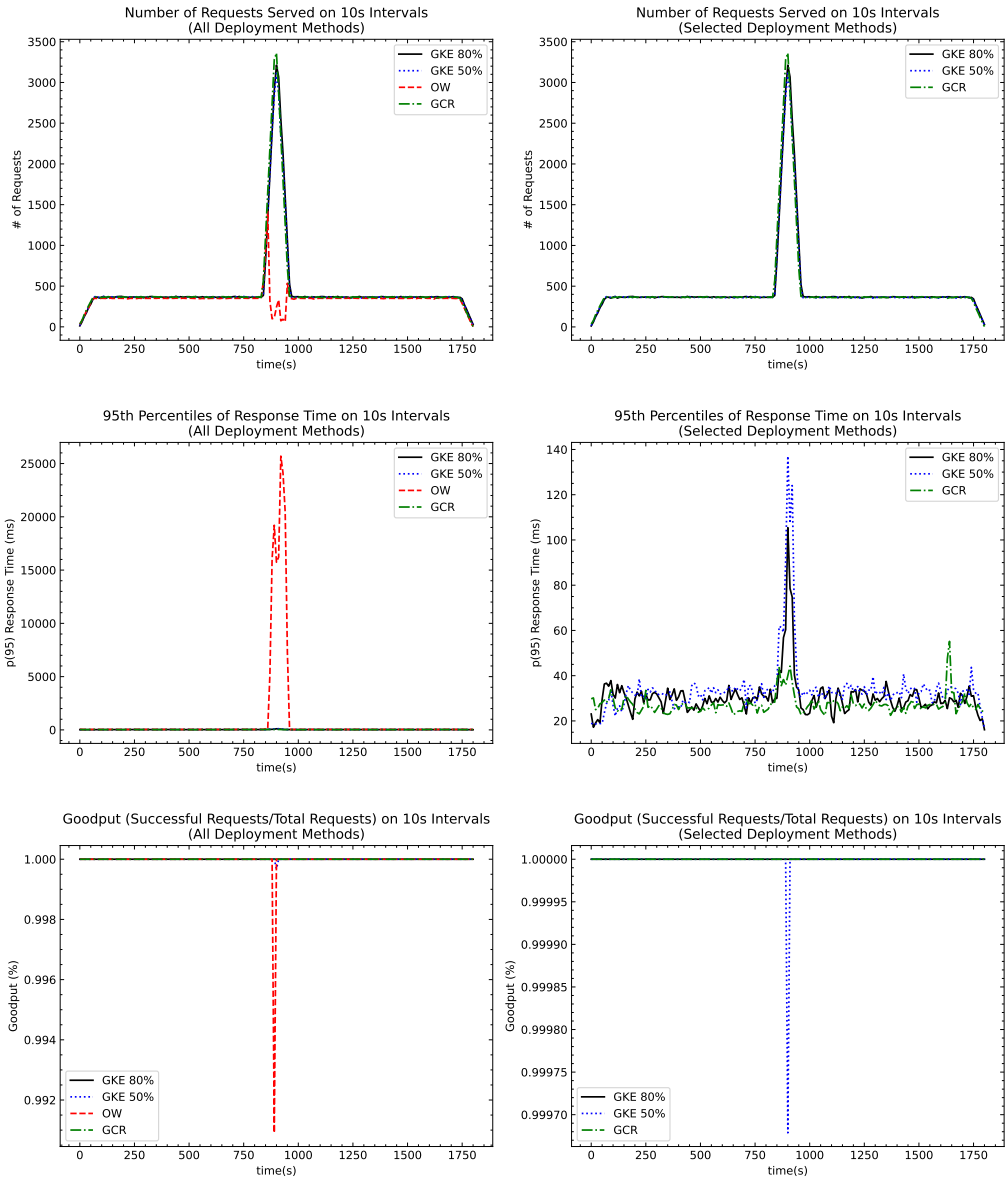


Figure 5.6.: Results of Spike Workload on `Users-Get` Endpoint

5.1.3. `Devices-Add` Endpoint

On the `Devices-Add` endpoint with a linear workload GCR served 310,090 requests with an average response time of 39.45ms, followed by GKE-50 (298,950 requests, 50.29ms) and GKE-80 (280,226 requests, 70.41ms). OW's response time again rapidly increased at 2,000 requests per 10s. In total it served 176,729 requests within 261.33ms on average. Notably is GCR's performance improvement in the last 5min of the experiment run. Being on par with GKE-50 for almost 25min of the experiment, the performance suddenly improved and response time was halved (c.f. Figure 5.7, middle row, right column).

Table 5.13.: Overall Request Counts of Linear Workload on `Devices-Add` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	1	298,950	100%
GKE-80	0	280,226	100%
OW	6	176,729	99,997%
GCR	0	310,090	100%

Table 5.14.: Overall Request Durations of Linear Workload on `Devices-Add` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	50.29ms	0s	48.55ms	372.01ms	85.82ms	93.35ms
GKE-80	70.41ms	13.57ms	75.74ms	378.89ms	115.81ms	125.9ms
OW	261.33ms	27.41ms	35.89ms	58.6s	48.52ms	66.4ms
GCR	39.45ms	19.64ms	30.08ms	2.21s	61.02ms	74.11ms

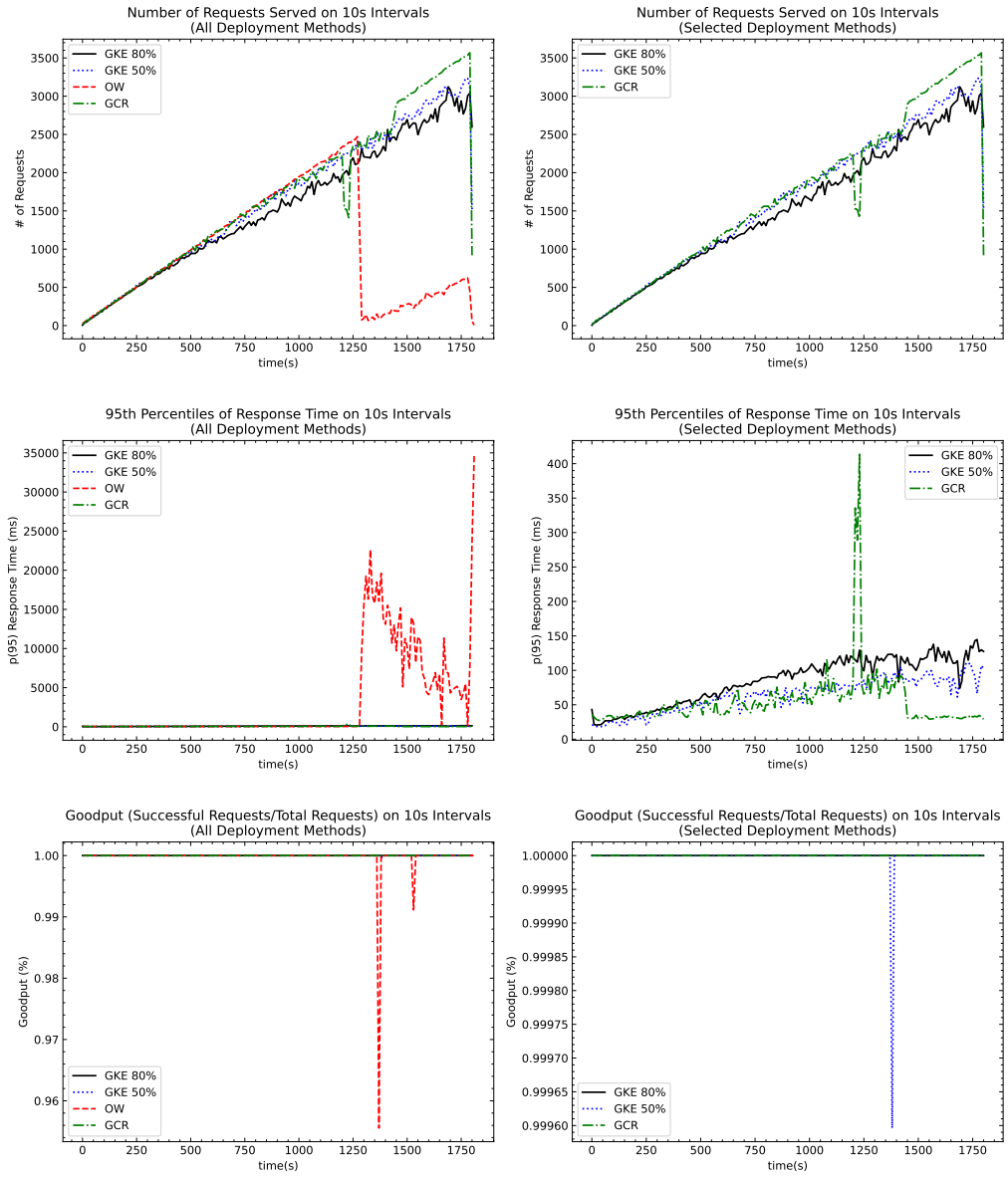


Figure 5.7.: Results of Linear Workload on `Devices-Add` Endpoint

The random workload resulted in 318,868served requests for GCR, 307,056 for GKE-50, 279,296 for GKE-80 and 166,670 for OW. The average response times were 34.2ms for GCR, 45.2ms for GKE-50, 74.61ms for GKE-80 and 296.9ms for OW. Again OW failed to compete with the other deployments when more than 2,000 requests were sent per 10s.

Table 5.15.: Overall Request Counts of Random Workload on `Devices-Add` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	0	307,056	100%
GKE-80	0	279,296	100%
OW	4	166,670	99.998%
GCR	1	318,868	100%

Table 5.16.: Overall Request Durations of Random Workload on `Devices-Add` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	45.2ms	13.17ms	48.44ms	285.4ms	71.2ms	76.89ms
GKE-80	74.61ms	13.75ms	76.37ms	359.14ms	124.36ms	135.03ms
OW	296.9ms	27.74ms	36.92ms	1m0s	55.58ms	78.76ms
GCR	34.2ms	9.84ms	27.77ms	2.53s	52.54ms	62.1ms

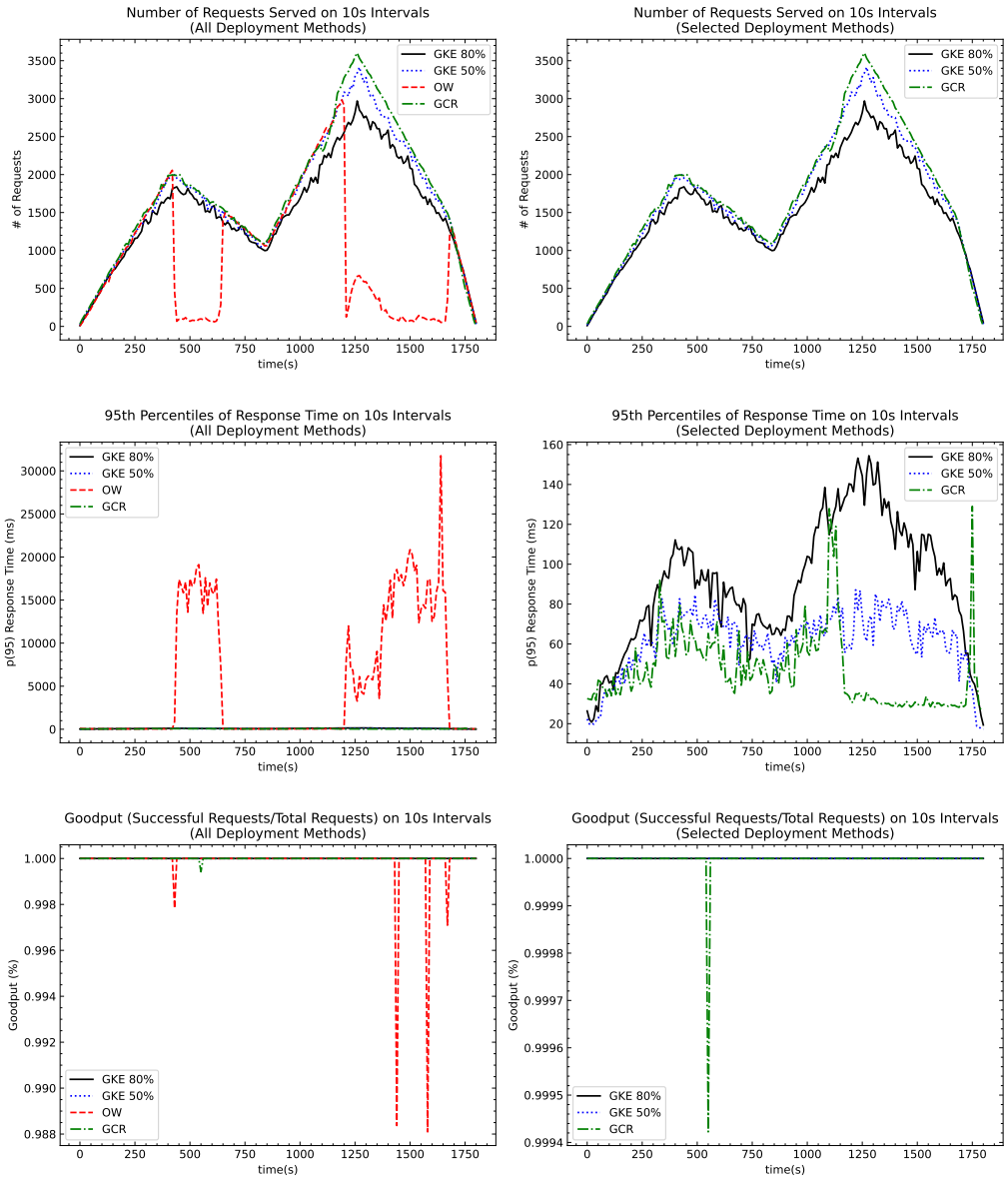


Figure 5.8.: Results of Random Workload on `Devices-Add` Endpoint

During the spike workload experiment run on the `Devices-Add` endpoint OW could catch up to some extent. GCR served 79,616 requests within 36.05ms on average, GKE-50 80,252 with an average of 33.83ms, GKE-80 78,108 with 41.68ms and OW 65,377 requests within 105.93 ms on average. This average is mainly increased by the substantially larger response times during the spike. OW's maximum response time was 27.97s, while the second largest maximum (GCR) was 1.8s

Table 5.17.: Overall Request Counts of Spike Workload on `Devices-Add` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	3	80,252	99.996%
GKE-80	0	78,108	100%
OW	1	65,377	100%
GCR	0	79,616	100%

Table 5.18.: Overall Request Durations of Spike Workload on `Devices-Add` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	33.83ms	0s	27.03ms	313.11ms	75.42ms	89.19ms
GKE-80	41.68ms	14.54ms	29.86ms	313.92ms	96.62ms	124.49ms
OW	105.43ms	28.41ms	36.94ms	27.97s	43.57ms	52.15ms
GCR	36.05ms	20.6ms	29.62ms	1.8s	53.26ms	72.29ms

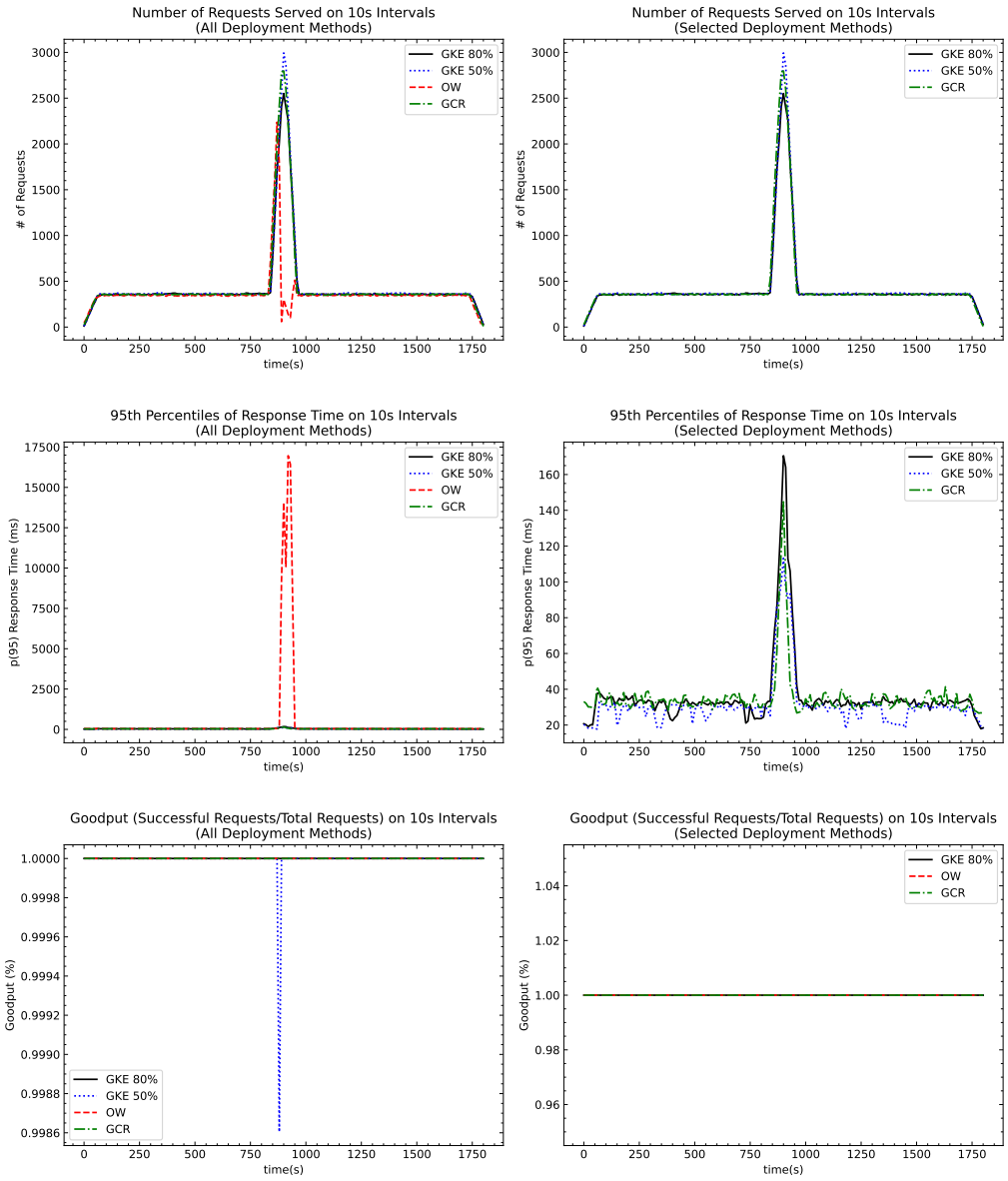


Figure 5.9.: Results of Spike Workload on `Devices-Add` Endpoint

5.1.4. `Devices-Get` Endpoint

On the `Devices-Get` endpoint the GKE deployments outperformed the serverless deployments. GKE-50 served 334,605 requests with an average response time of 18.18ms, GKE-80 served 334,605 requests with an average response time of 19.75ms. GCR followed with a difference of more than 10,000 requests (322,234) and an average response time of 28.48ms. OW could only serve 227,558 requests within 146.35ms on average.

Table 5.19.: Overall Request Counts of Linear Workload on `Devices-Get` Endpoint

Deployment Method	Failed	Total	Goodput
GKE-50	0	334,605	100%
GKE-80	0	332,665	100%
OW	5	227,558	99,998%
GCR	0	322,234	100%

Table 5.20.: Overall Request Durations of Linear Workload on `Devices-Get` Endpoint

Deployment Method	AVG	MIN	MED	MAX	p(90)	p(95)
GKE-50	18.18ms	14.19ms	17.24ms	139.09ms	21.05ms	23.65ms
GKE-80	19.75ms	14.41ms	17.91ms	155.99ms	24.3ms	29.13ms
OW	146.35ms	28.21ms	40.34ms	33.24s	63.42ms	76.99ms
GCR	28.48ms	20.31ms	26.83ms	2.11s	34.63ms	39.29ms

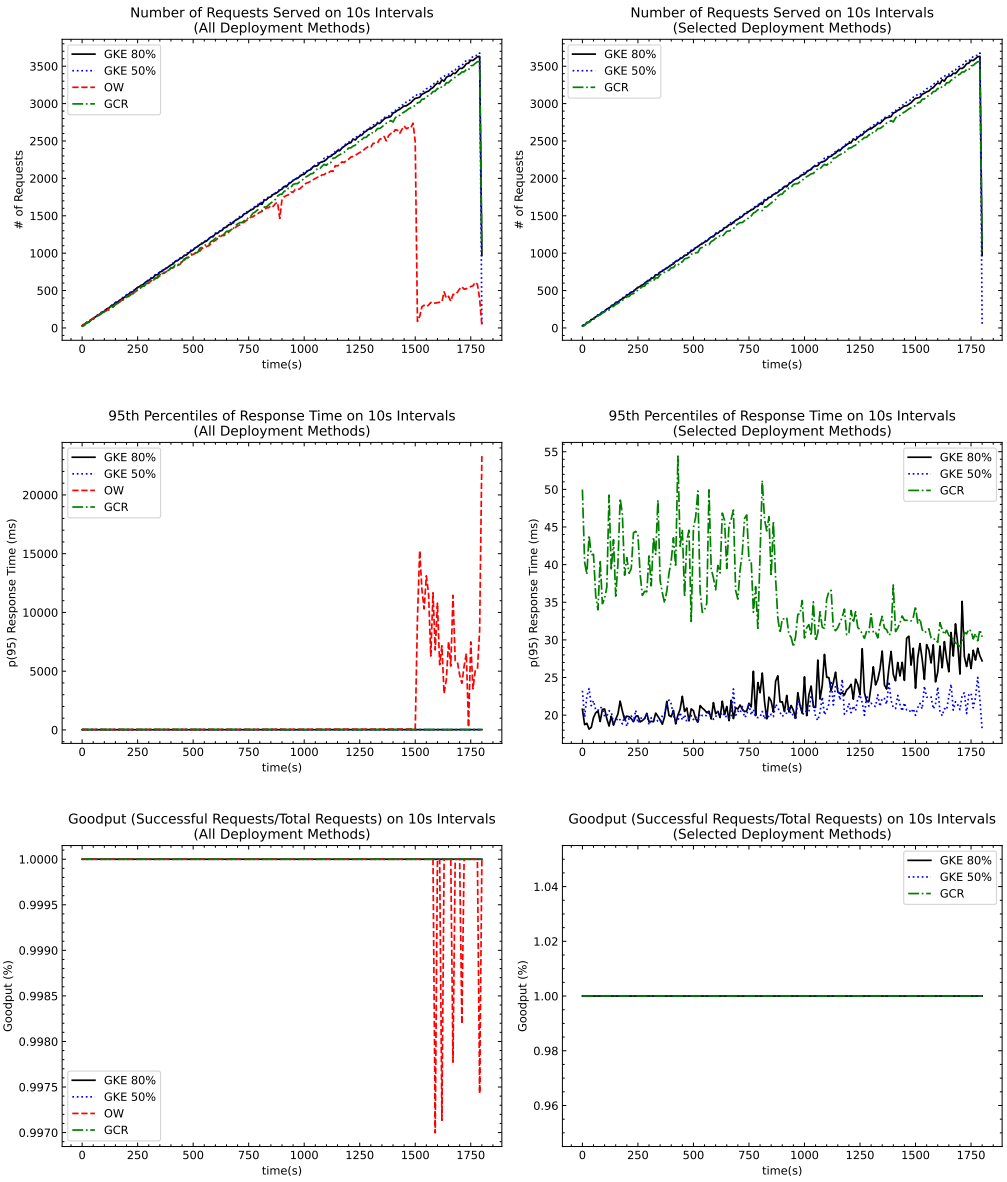


Figure 5.10.: Results of Linear Workload on `Devices-Get` Endpoint

The random workload replicated the impression of the linear workload. GKE-50 served 337,044 requests in total and the average response time was 18.81ms. GKE-80 followed with 335,255 requests and an average response time of 20.26ms. GCR served 324,856 requests within 28.93ms on average. OW followed with a substantial distance of more than 160,000 requests (162,407 requests served in total) and an average response time of 310.86ms.

Table 5.21.: Overall Request Counts of Random Workload on `Devices-Get` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	0	337,044	100%
GKE-80	0	335,255	100%
OW	6	162,407	99.996%
GCR	2	324,856	99.999%

Table 5.22.: Overall Request Durations of Random Workload on `Devices-Get` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	18.81ms	14.51ms	17.78ms	241.64ms	21.8ms	24.49ms
GKE-80	20.26ms	14.81ms	18.76ms	264.59ms	24.5ms	28.4ms
OW	310.86ms	28.27ms	40.41ms	30.3s	65.72ms	101.04ms
GCR	28.93ms	9.81ms	26.22ms	14.65s	32.51ms	37.55ms

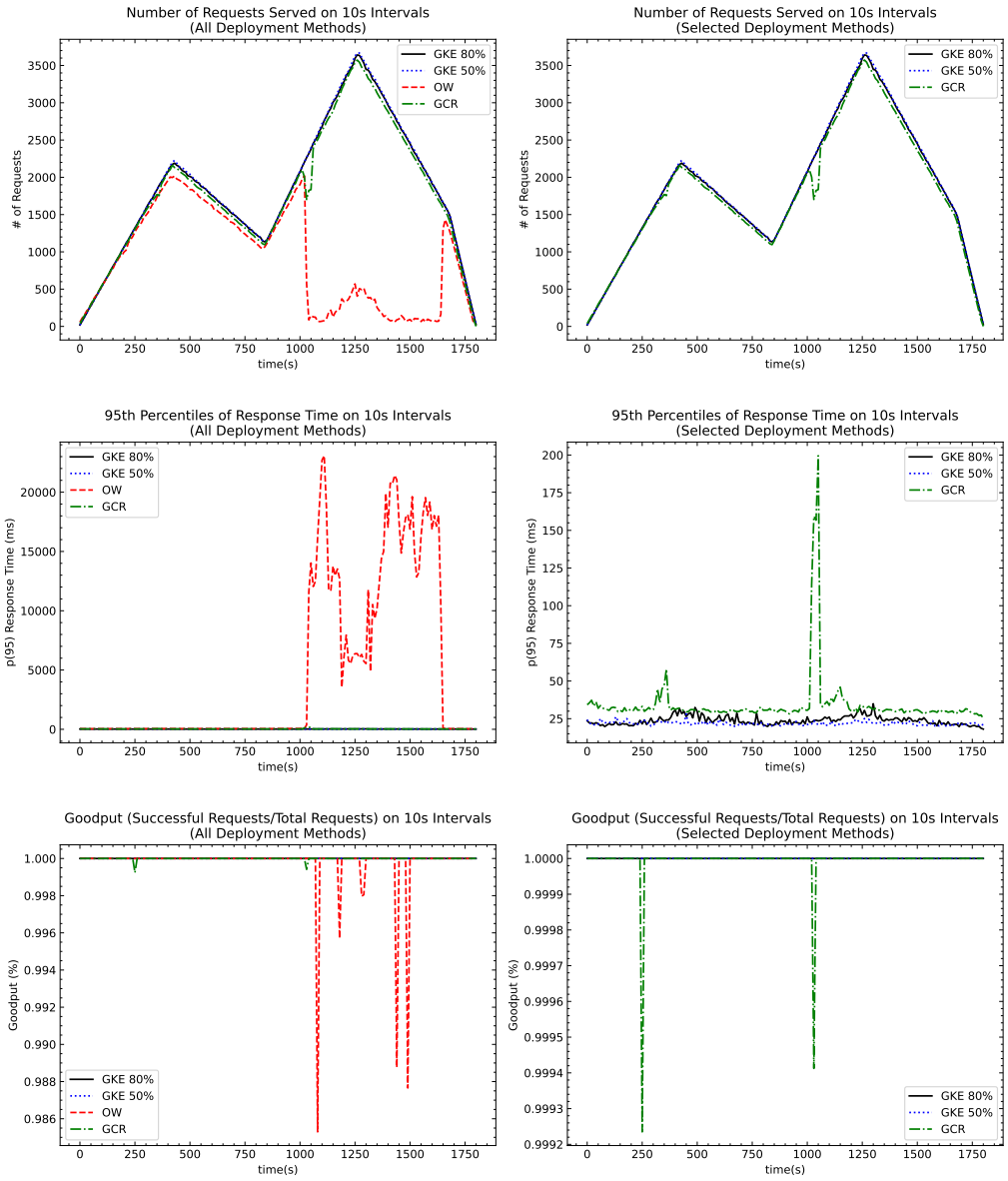


Figure 5.11.: Results of Random Workload on `Devices-Get` Endpoint

During the spike load experiment run GKE-80 served 83,207 requests (avg. response time: 23.64ms), GKE-50 80,429 requests (33.17ms), GCR 79,616 requests (36.05ms) and OW 62,137 requests (125.25ms). As for previous spike workloads OW was not able to scale as quickly as GCR during the sudden increase in workload. The maximum response time rose to 33.55s in comparison to 1.8s for GCR.

Table 5.23.: Overall Request Counts of Spike Workload on `Devices-Get` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	0	80,429	100%
GKE-80	0	83,207	100%
OW	0	62,137	100%
GCR	0	79,616	100%

Table 5.24.: Overall Request Durations of Spike Workload on `Devices-Get` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	33.17ms	15.37ms	19.75ms	370.92ms	66.55ms	131.56ms
GKE-80	23.64ms	14.63ms	17.69ms	216.18ms	36.21ms	62.57ms
OW	125.25ms	28.06ms	37.01ms	33.55s	44.17ms	52.14ms
GCR	36.05ms	20.6ms	29.62ms	1.8s	53.26ms	72.29ms

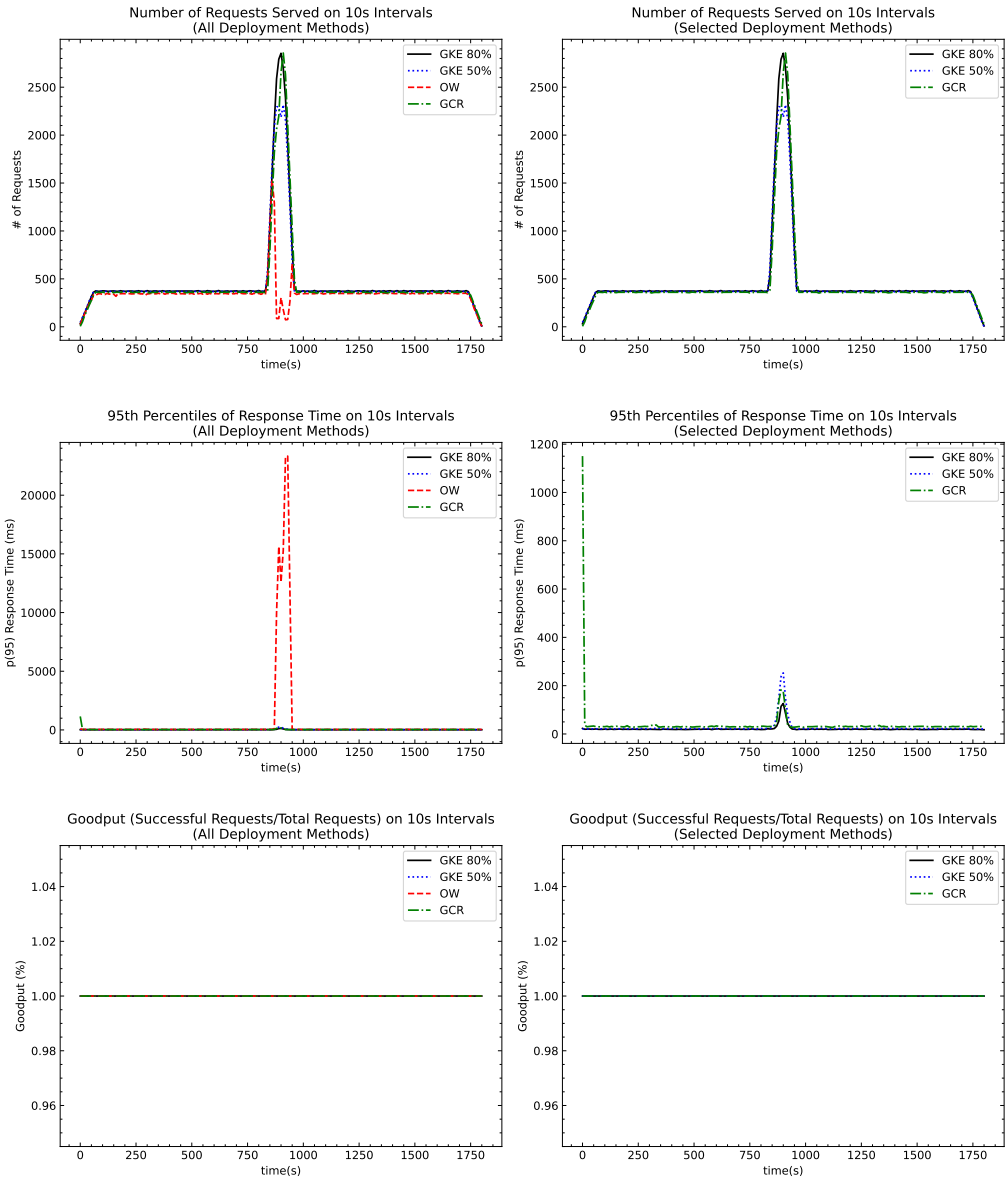


Figure 5.12.: Results of Spike Workload on `Devices-Get` Endpoint

5.1.5. `Sensors-Add` Endpoint

The `Sensors-Add` endpoint experiments did not run as intended. As already stated at the beginning of this chapter, two services of the shared backend, Elasticsearch and Kafka, could not serve the high workloads. The therefore strongly shortened experiment runs of 5min still revealed interesting insights. The numbers of requests served deviated substantially between microservice and FaaS implementations. It is based on a simple architectural advantage the microservice architecture has. The IoT core creates the sensor object on MariaDB and returns a status 200 to the user. In parallel but not at the same time, it triggers three requests to Elasticsearch, Kafka and Connect to create a new index, topic and connect job. The overall response is independent of the results of these requests. This functionality is based upon the assumption that the IoT core instance is continuously running after serving a request. This is where a FaaS function deviates. It has to wait for the responses, because it has to count in the lifespan of a function Docker container. It is likely that one or more requests are not even sent anymore as container instances can be shut down immediately after sending a response. Waiting for overburdened Elasticsearch and Kafka explains how OW and GCR could only serve 698 and 664 requests, while GKE-50 and GKE-80 served 21,900 and 38,535 requests.

Table 5.25.: Overall Request Counts of Sprint Workload on `Sensors-Add` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	0	21,900	100%
GKE-80	0	38,535	100%
OW	76	698	89.112%
GCR	206	664	68.976%

Table 5.26.: Overall Request Durations of Sprint Workload on `Sensors-Add` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	445.43ms	19.04ms	64.86ms	1m0s	183.11ms	262.29ms
GKE-80	137.48ms	19.86ms	115.66ms	5.31s	208.74ms	263.88ms
OW	22.9s	1.28s	21.18s	1m0s	44.11s	49.38s
GCR	22.02s	784.01ms	6.31s	1m0s	1m0s	1m0s

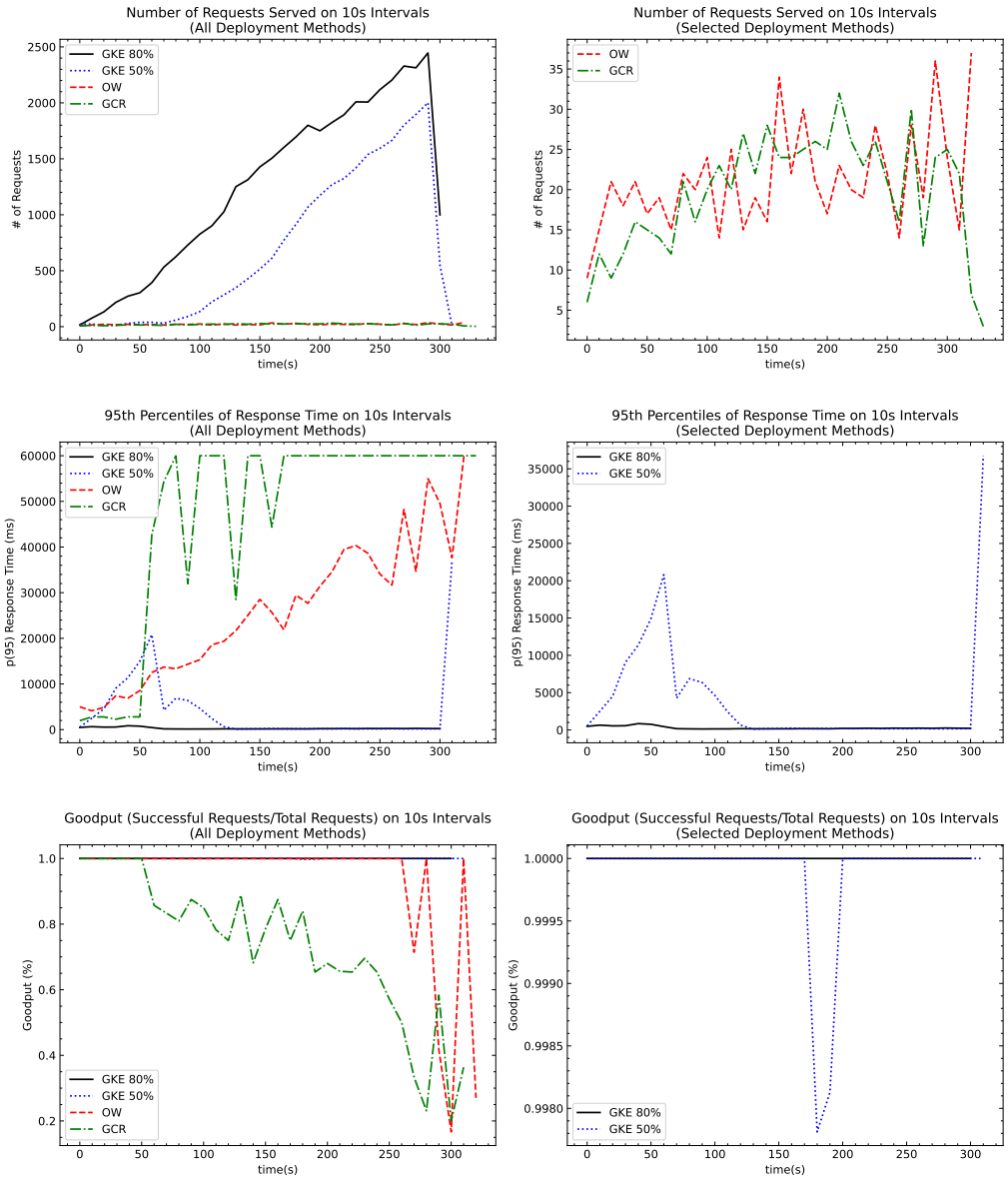


Figure 5.13.: Results of Sprint Workload on `Sensors-Add` Endpoint

5.1.6. `Sensors-Get` Endpoint

The `Sensors-Get` endpoint results resemble more closely what has been witnessed on the previous experiment runs. For the linear workload GKE-80 was able to serve 343,343 requests with an average response time of 11.35ms, GKE-50 served 343,158 requests within 11.47ms on average, GCR served 319,046 requests within 31.29ms on average and OW served 161,053 requests within 310.08ms on average. OW's previously observed threshold of 2,000 requests per 10s occurred in this run again.

Table 5.27.: Overall Request Counts of Linear Workload on `Sensors-Get` Endpoint

Deployment Method	Failed	Total	Goodput
GKE-50	0	343,158	100%
GKE-80	0	343,343	100%
OW	7	161,053	99,996%
GCR	0	319,046	100%

Table 5.28.: Overall Request Durations of Linear Workload on `Sensors-Get` Endpoint

Deployment Method	AVG	MIN	MED	MAX	p(90)	p(95)
GKE-50	11.47ms	9.81ms	10.9ms	237.71ms	11.84ms	19.72ms
GKE-80	11.35ms	9.76ms	10.78ms	65.41ms	11.71ms	19.63ms
OW	310.08ms	27.21ms	38.08ms	31.6s	64.23ms	94.15ms
GCR	31.29ms	19.46ms	26.88ms	2.28s	46.56ms	53.74ms

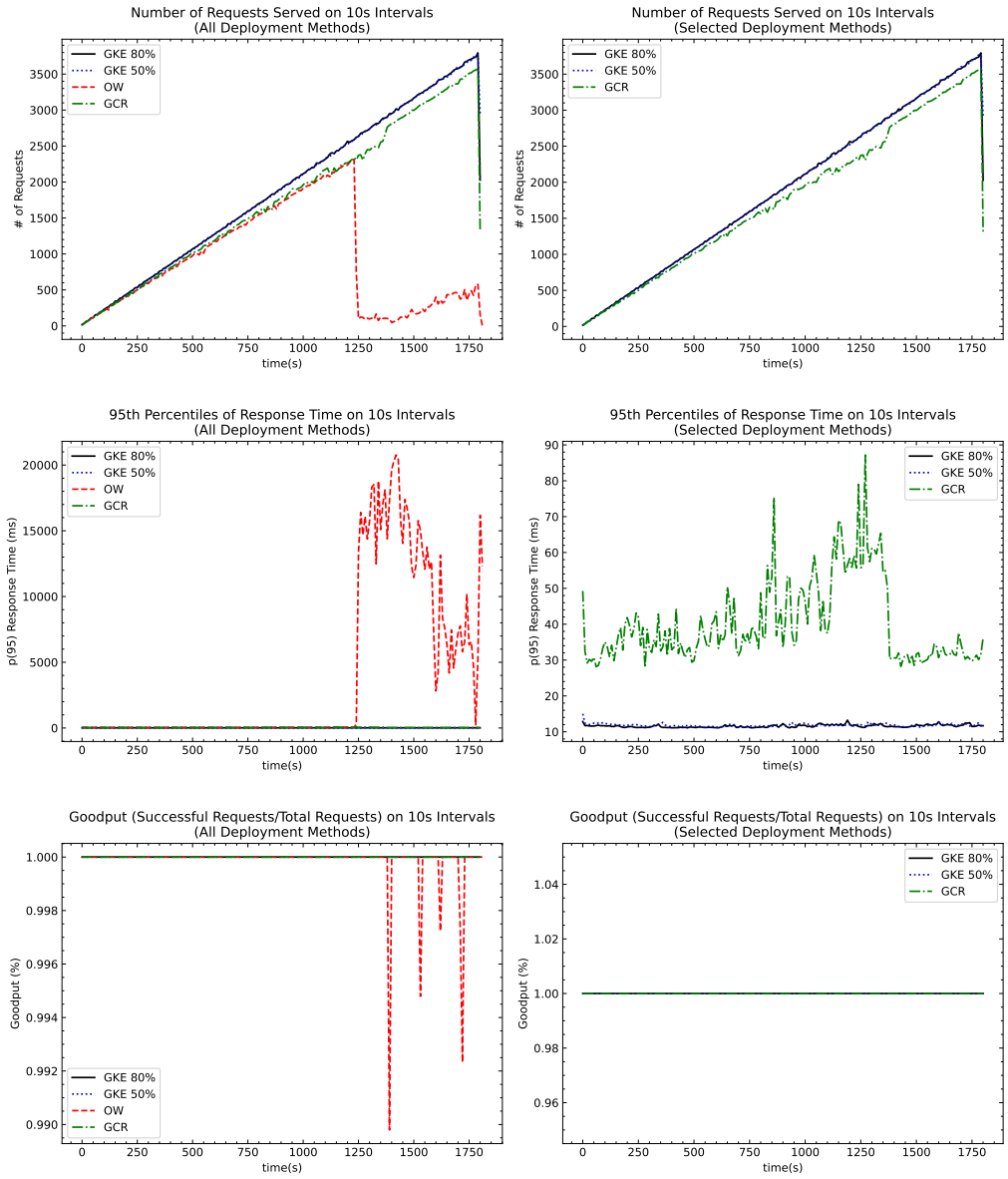


Figure 5.14.: Results of Linear Workload on `Sensors-Get` Endpoint

For the random workload GKE-50 and GKE-80 could answer 346,864 and 346,620 requests within 11.18ms and 11.39ms on average respectively. GCR was able to respond within 31.45ms on average and served a total of 321,963 requests. OW answered only 175,065 requests with an average response time of 267.47ms.

Table 5.29.: Overall Request Counts of Random Workload on `Sensors-Get` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	0	346,864	100%
GKE-80	0	346,620	100%
OW	9	176,065	99.995%
GCR	0	321,963	100%

Table 5.30.: Overall Request Durations of Random Workload on `Sensors-Get` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	11.18ms	9.78ms	10.64ms	53.85ms	11.33ms	19.54ms
GKE-80	11.39ms	9.83ms	10.82ms	44.2ms	11.83ms	19.65ms
OW	267.47ms	27.46ms	37.68ms	35.52s	62.34ms	86.75ms
GCR	31.45ms	19.71ms	27.43ms	2.07s	44.71ms	52.86ms

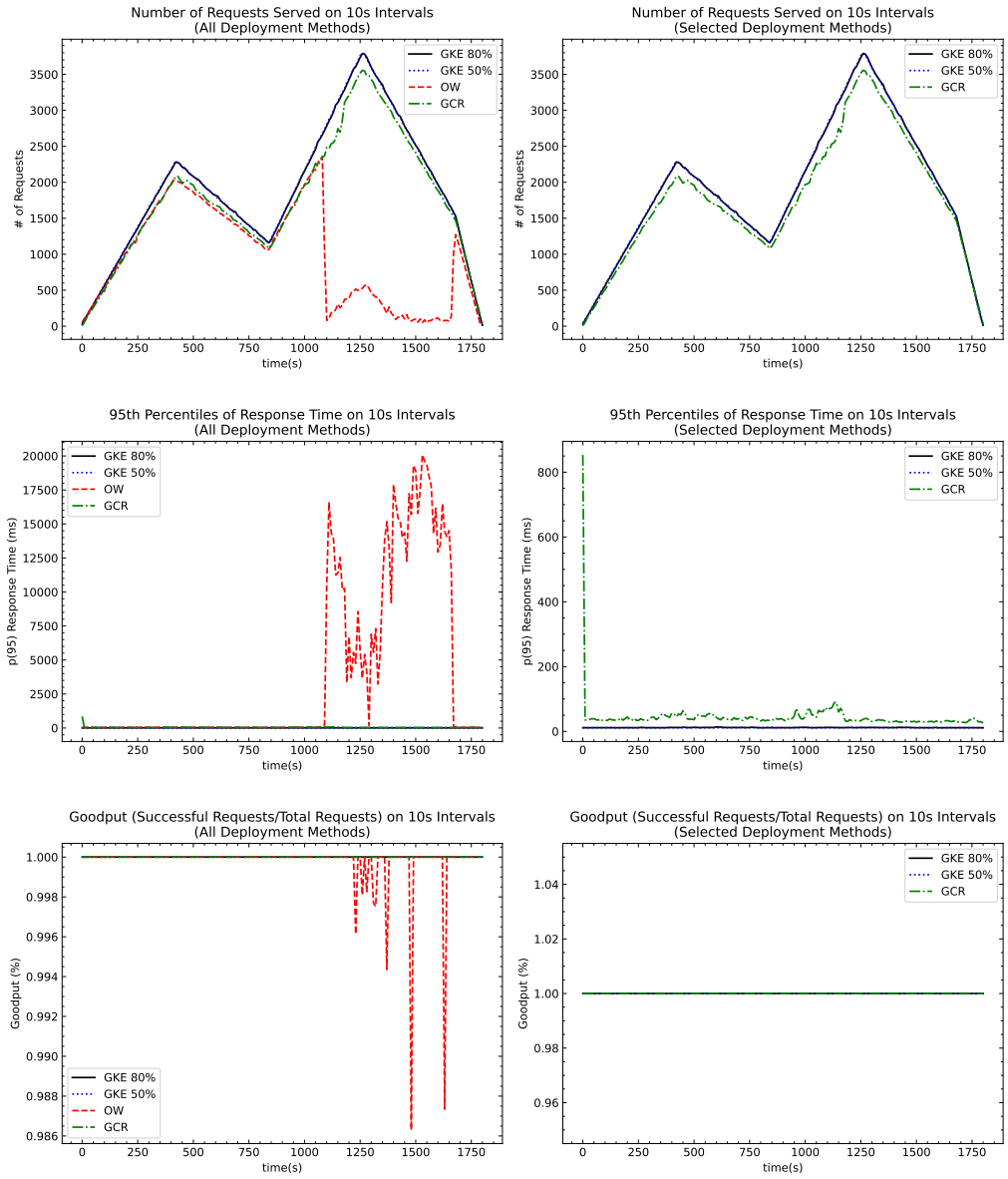


Figure 5.15.: Results of Random Workload on `Sensors-Get` Endpoint

The spike workload completed the tests on the `Sensors-Get` endpoint. GKE-50 served a total of 87,132 requests, GKE-80 87,085, GCR 79,628 and OW 68,409. The average response times were 11.24ms, 11.38ms, 36.05ms and 88.8ms.

Table 5.31.: Overall Request Counts of Spike Workload on `Sensors-Get` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	0	87,132	100%
GKE-80	0	87,085	100%
OW	3	68,409	99.996%
GCR	0	79,628	100%

Table 5.32.: Overall Request Durations of Spike Workload on `Sensors-Get` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	11.24ms	9.9ms	10.74ms	39.26ms	11.26ms	19.59ms
GKE-80	11.38ms	9.91ms	10.9ms	41.8ms	11.31ms	19.76ms
OW	88.8ms	28.72ms	36.97ms	29.33s	47.86ms	65.11ms
GCR	36.05ms	20.31ms	30.49ms	1.98s	50.82ms	69.73ms

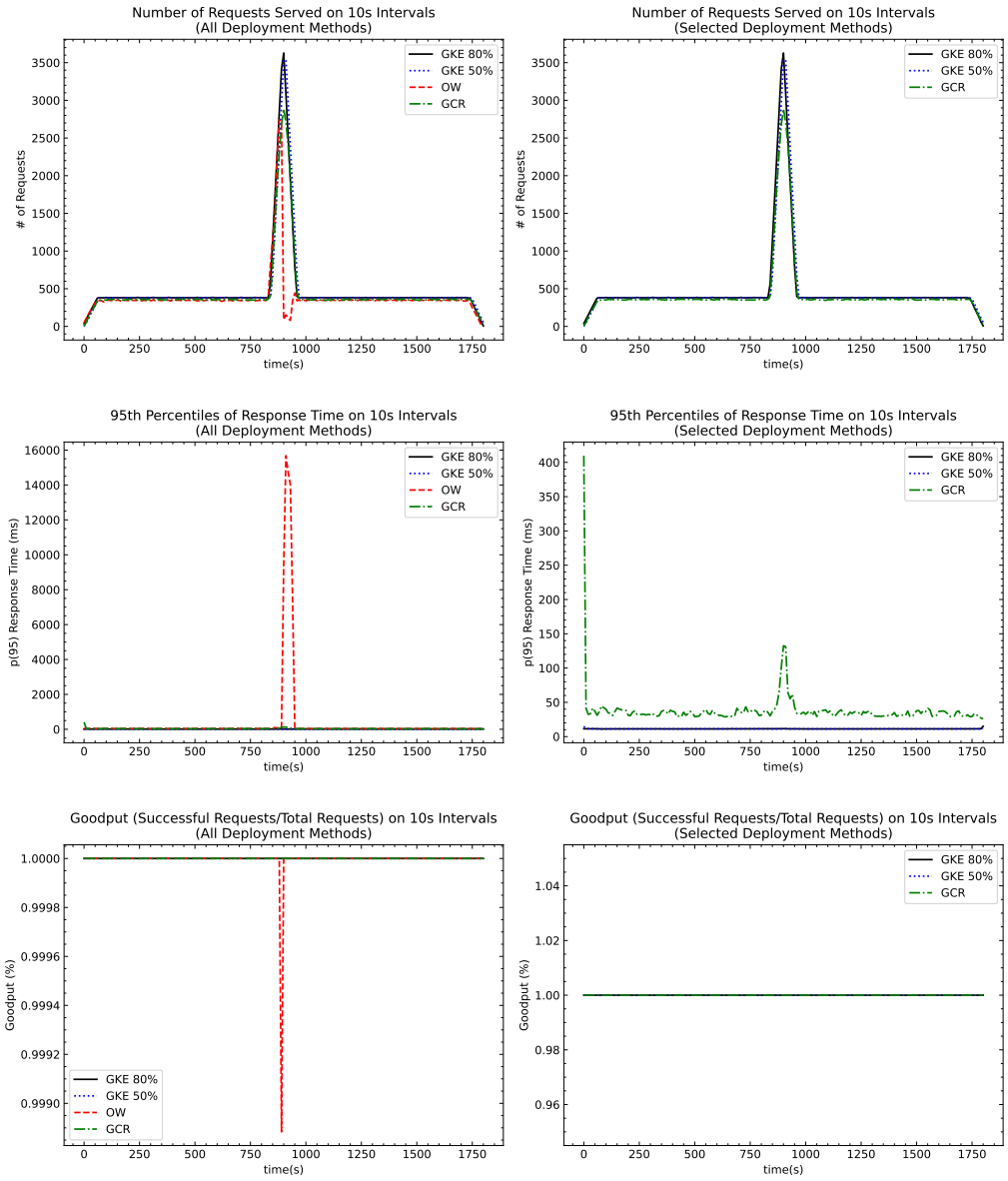


Figure 5.16.: Results of Spike Workload on `Sensors-Get` Endpoint

5.1.7. HTTP-Gateway

The experiments on the HTTP-Gateway with the linear workload displayed again results, in which the serverless deployments could rival and even exceed the performance of the serverful deployments. OW served 322,721 requests and reached an average response time of 28.02ms. For GCR these numbers were 299,420 and 49.73ms. GKE-50 and GKE-80 followed with a small distance. Former served 295,154 requests within 54.16ms on average, latter 283,433 requests within 66.82ms.

Table 5.33.: Overall Request Counts of Linear Workload on HTTP-Gateway Endpoint

Deployment Method	Failed	Total	Goodput
GKE-50	0	295,154	100%
GKE-80	0	283,433	100%
OW	0	322,721	100%
GCR	9	299,420	99.997%

Table 5.34.: Overall Request Durations of Linear Workload on HTTP-Gateway Endpoint

Deployment Method	AVG	MIN	MED	MAX	p(90)	p(95)
GKE-50	54.16ms	13.7ms	47.61ms	968.36ms	96.93ms	113.41ms
GKE-80	66.82ms	13.59ms	66.18ms	392.86ms	108.63ms	123.64ms
OW	28.02ms	19.27ms	25.66ms	4.11s	33.35ms	39.33ms
GCR	49.73ms	9.94ms	40.93ms	11.07s	88.56ms	108.39ms

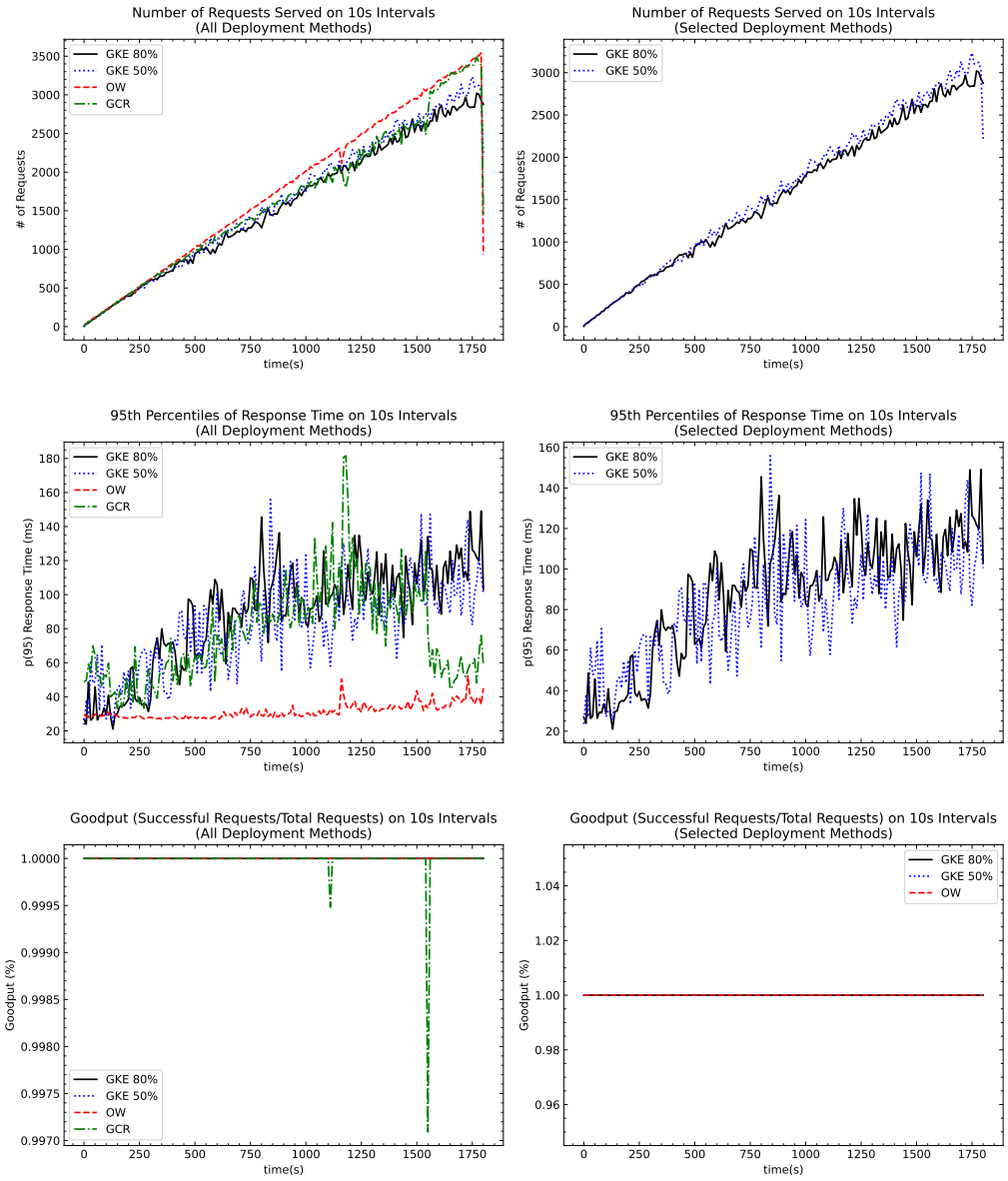


Figure 5.17.: Results of Linear Workload on `HTTP-Gateway`

The random workload had a similar outcome as the linear workload. The serverless deployments outperformed the serverful ones. OW served 326,900 requests, GCR 311,514, GKE-50 297,088 and GKE-80 283,071. The average response times were 27.15ms, 40.88ms, 55.11ms and 70.29ms respectively.

Table 5.35.: Overall Request Counts of Random Workload on `HTTP-Gateway` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	1	297,088	100%
GKE-80	0	283,071	100%
OW	0	326,900	100%
GCR	4	311,514	99.999%

Table 5.36.: Overall Request Durations of Random Workload on `HTTP-Gateway` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	55.11ms	0s	51.97ms	269.83ms	95.42ms	109.61ms
GKE-80	70.29ms	13.34ms	69.37ms	384.8ms	113.54ms	130.61ms
OW	27.15ms	19.77ms	24.86ms	3.9s	31.81ms	37.63ms
GCR	40.88ms	9.86ms	30.89ms	19.63s	70.82ms	88.38ms

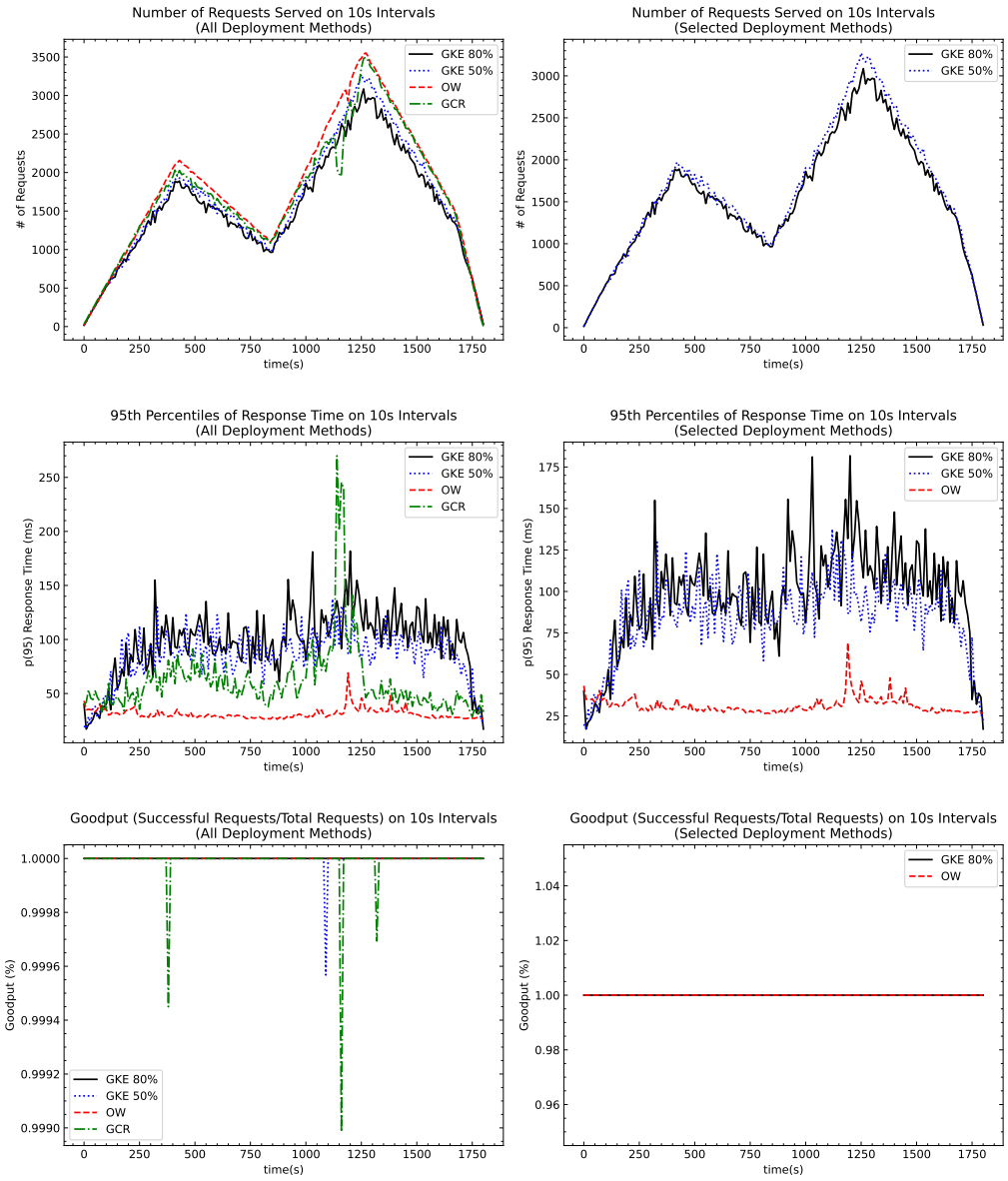


Figure 5.18.: Results of Random Workload on `HTTP-Gateway`

The spike workload resulted in the following metrics. OW served 81,971 requests and reached an average response time of 27.74ms, GCR served 81,029 requests within 30.98ms on average, GKE-80 answered 76,256 requests with an average response time of 48.76ms and GKE-50 served 75,969 requests within 49.87ms on average.

Table 5.37.: Overall Request Counts of Spike Workload on `HTTP-Gateway` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	0	75,969	100%
GKE-80	0	76,256	100%
OW	0	81,971	100%
GCR	0	81,029	100%

Table 5.38.: Overall Request Durations of Spike Workload on `HTTP-Gateway` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	49.87ms	14.15ms	31.55ms	524.5ms	116.55ms	148.34ms
GKE-80	48.76ms	14.27ms	29.87ms	532.31ms	125.16ms	152.55ms
OW	27.74ms	20.25ms	25.13ms	4.84s	29.55ms	33.61ms
GCR	30.98ms	17ms	22.47ms	1.59s	57.56ms	72.45ms

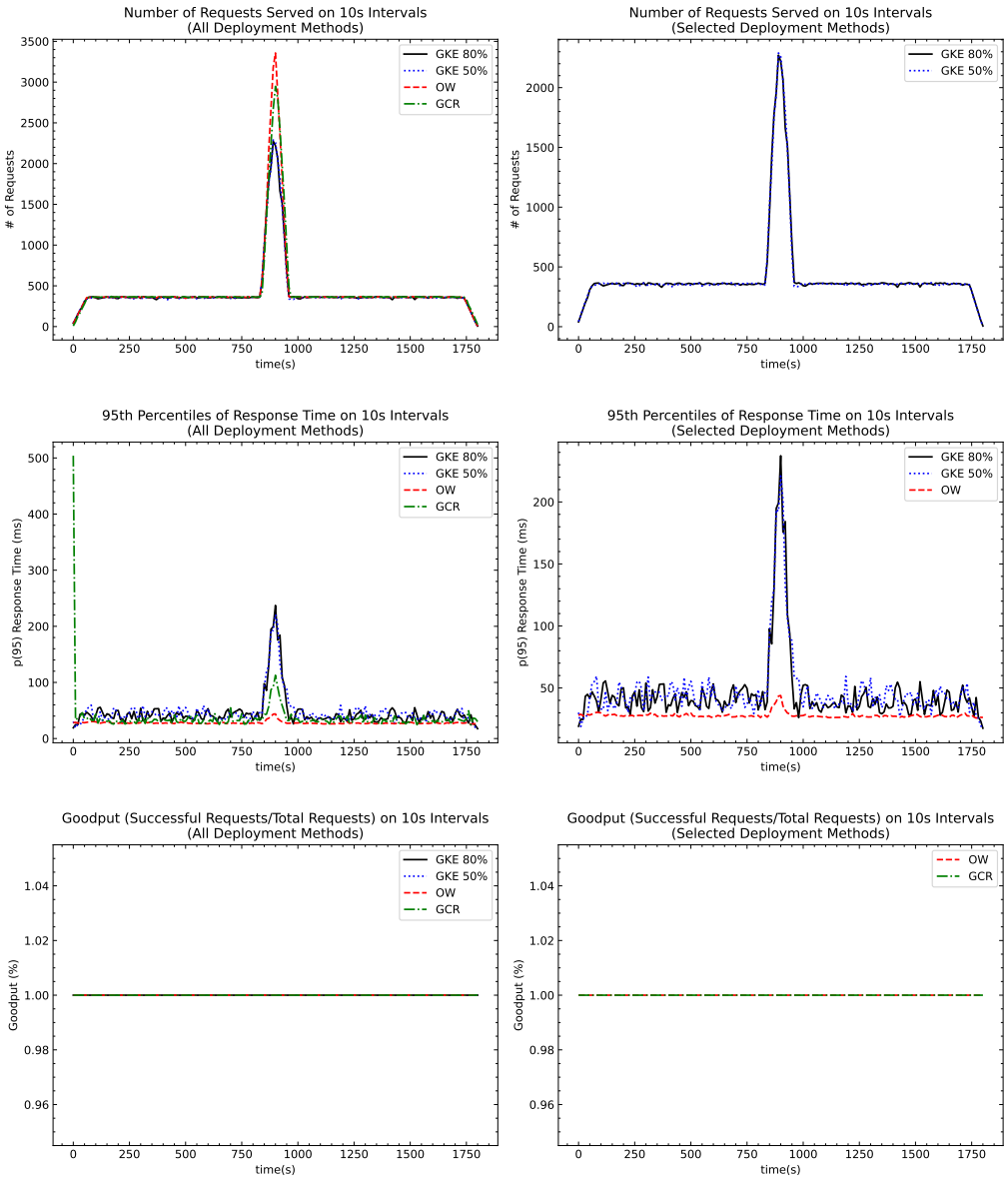


Figure 5.19.: Results of Spike Workload on `HTTP-Gateway`

5.1.8. `Consumers-Consume-Get` Endpoint

The results on the `Consumers-Consume-Get` endpoint were arranged closely together with OW serving 314,545 requests, GKE-50 answering 312,580 requests, GKE-80 serving 311,265 requests and GCR answering 307,828 requests. Average response times were also distributed in a small range. OW reached 35.31ms, GKE-50 37.18ms, GKE-80 38.37ms and GCR 41.56ms. The GCR deployment sticks out with a very high failure rate, the goodput fell to 70.887%. It is likely that this is caused by issues within the VPC connector. The goodput drops in periodical time intervals, recovers slowly and then drops again. This endpoint requires the highest bandwidth of all endpoints for the data retrieval from Elasticsearch. For the deployment only the smallest VM instances were used to connect the GCR service with the shared services on the VPC. Though they could scale to 10 instances, maybe the step increments were too small to serve the traffic.

Table 5.39.: Overall Request Counts of Linear Workload on `Consumers-Consume-Get` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	1	312,530	100%
GKE-80	0	311,265	100%
OW	0	314,545	100%
GCR	89,617	307,828	70.887%

Table 5.40.: Overall Request Durations of Linear Workload on `Consumers-Consume-Get` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	37.18ms	0s	28.54ms	1.03s	64.7ms	83.18ms
GKE-80	38.37ms	15.72ms	29.83ms	515.04ms	66.76ms	86.08ms
OW	35.31ms	23.74ms	32.22ms	4.98s	42.6ms	52.23ms
GCR	41.56ms	18.75ms	33.85ms	2.6s	67.48ms	86.52ms

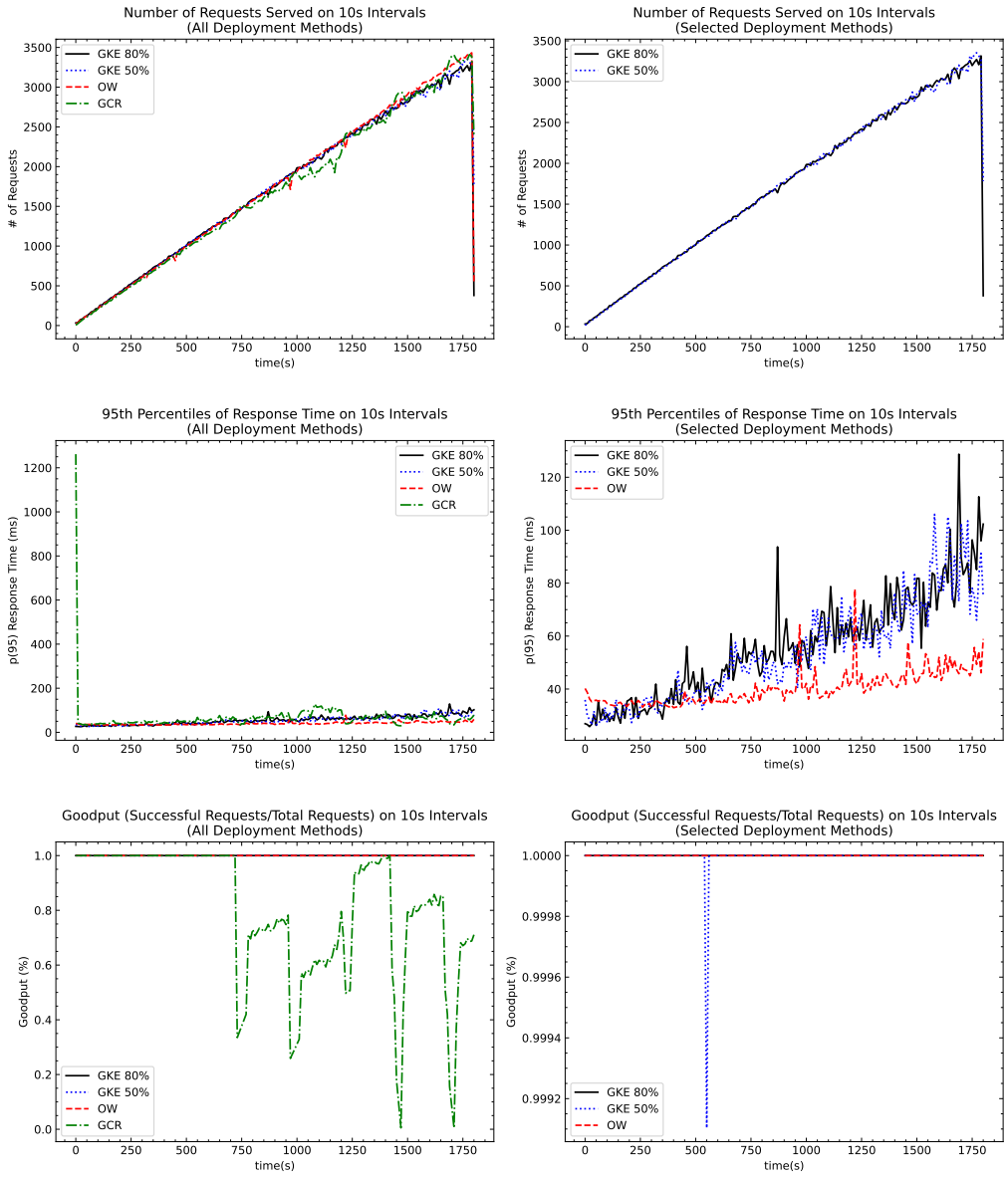


Figure 5.20.: Results of Linear Workload on `Consumers-Consume-Get` Endpoint

The observations from the line workload are replicated with the random workload. While according to requests served (GKE-50: 318,553, OW: 317,326, GKE-80: 316,838, GCR: 312,235) and average response time (GKE-50: 34.48ms, OW: 36.02ms, GKE-80: 35.57ms, GCR: 40.23ms) the deployments performed quite similar, the goodput sets GCR apart. OVerall it only answered 72.246% of the requests successfully. Again the hypothesis identifies the VPC connector as the likely origin of this failure.

Table 5.41.: Overall Request Counts of Random Workload on `Consumers-Consume-Get` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	0	318,553	100%
GKE-80	0	316,838	100%
OW	0	317,326	100%
GCR	86,658	312,235	72.246%

Table 5.42.: Overall Request Durations of Random Workload on `Consumers-Consume-Get` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	34.48ms	15.82ms	27.04ms	3.11s	57.05ms	72.17ms
GKE-80	36.02ms	15.9ms	28.22ms	830ms	60.97ms	77.74ms
OW	35.57ms	24.09ms	32.21ms	5.71s	41.95ms	51.52ms
GCR	40.23ms	18.46ms	33.3ms	2.75s	63.95ms	81.38ms

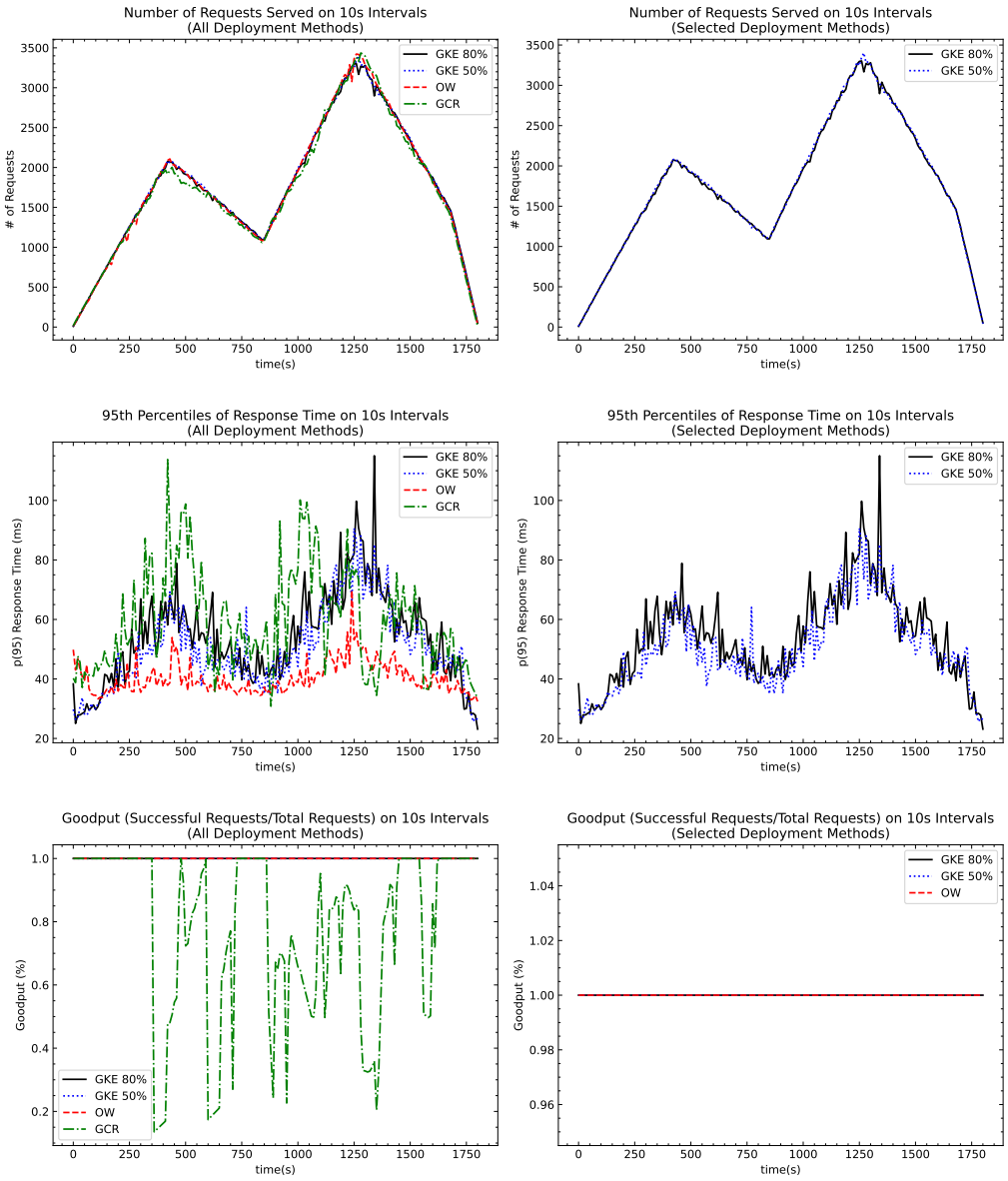


Figure 5.21.: Results of Random Workload on `Consumers-Consume-Get` Endpoint

The final spike load experiment runs on this endpoint did not reveal any substantial failures as the previous two workloads have shown. GKE-50 served 81,593 requests in total (avg. response time: 29.07ms), GKE-80 81,478 requests (29.45ms), OW 79,473 requests (36.53ms) and GCR 77,493 (43.85ms).

Table 5.43.: Overall Request Counts of Spike Workload on `Consumers-Consume-Get` Endpoint

Deployment Method	**Failed**	**Total**	**Goodput**
GKE-50	2	81,593	99.998%
GKE-80	0	81,478	100%
OW	0	79,473	100%
GCR	0	77,493	100%

Table 5.44.: Overall Request Durations of Spike Workload on `Consumers-Consume-Get` Endpoint

Deployment Method	**AVG**	**MIN**	**MED**	**MAX**	**p(90)**	**p(95)**
GKE-50	29.07ms	0s	23.36ms	251.1ms	46.72ms	61.77ms
GKE-80	29.45ms	16.1ms	23.47ms	272.79ms	46.76ms	63.27ms
OW	36.53ms	24.39ms	32.05ms	6.65s	38.67ms	45.68ms
GCR	43.85ms	22.36ms	32.25ms	2.19s	74.15ms	115.31ms

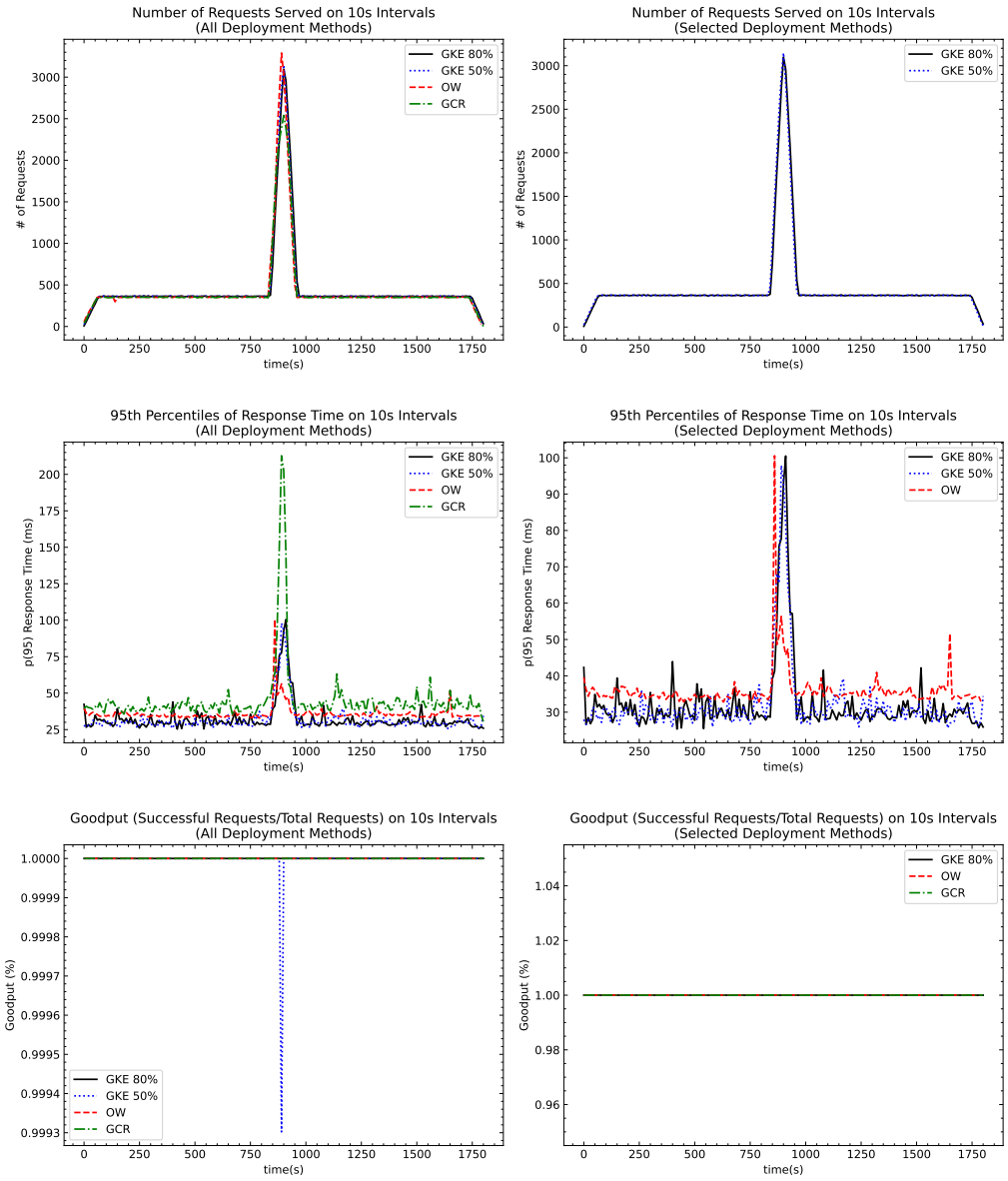


Figure 5.22.: Results of Spike Workload on `Consumers-Consume-Get` Endpoint

5.2. Cost Analysis

The purpose of this study was not only to compare performance of the four different deployment methods, but also to evaluate how costs differ for each of them. The starting point for the cost analysis was the overall cost breakdown of the invoice month April, 2022, of the whole Google Cloud project. As prices on the price sheet from the Google Cloud are listed in rather coarse measurements of hours, sometimes even months, and furthermore are only stated in USD, the first step was to calculate more accurate costs in EUR for one vCPU, one GB of either RAM or disk memory, GKE management fees etcetera per second, which is the unit of billing in the monthly cloud invoice.

Tracing back from the actual cost listed in the breakdown, a node of 8 vCPUs, 32 GB of RAM and 100 GB of persistent disk costs $8.7959 * 10^{-3}$ct per second in the location of Europe-West-3 in Frankfurt. The cost of running a 30min experiment on both GKE and OW splits up into the cost of the nodes deployed plus the GKE management fee of $2.4996 * 10^{-3}$ct per second. As the OW experiment setup was fixed to five nodes, total costs summed up to 83.6619ct for a half an hour long experiment run. The GKE clusters were cheaper, as they started with two nodes into an experiment run and only scaled according to demand. The base cost were 36.1643ct for a 30min experiment run. In seven occasions the cluster scaled to 3 nodes, which was added to the cost accordingly. For the Users-Signin endpoint, the GKE-50 setup scaled to three nodes for 2,288s (linear workload), for 1,565s (random) and for 830s (spike). GKE-80 scaled to three nodes for 1,126s (linear) and 1,316s (random). While testing the sprint test pattern on the Sensors-Add endpoint the GKE-50 scaled to three nodes for 129s and GKE-80 for 19s.

The cost of running a GCR service cannot be calculated that precisely. In general one pays for the number of invocations plus the vCPU seconds and RAM seconds used for that invocation. As a fixed amount of invocations is free, the cost were averaged over all invocations of the invoice month. Additionally, it was not traceable how many vCPU seconds or RAM seconds a single invocation caused. This was approximated with the average response time in an experiment run. The total cost of vCPU and RAM seconds billed in the invoice month were divided by the sum of GCR request multiplied by their respective average response time over all experiment runs. vCPU and RAM cost were then distributed to each experiment run by multiplying the previously computed value with the product of requests served times the average response time. Total costs of all experiment runs can be found in the Appendix (c.f. Table A.1). The following Table 5.45 displays the cost per 1,000 requests served in each experiment run for all four deployment methods.

The table shows that very consistently the GCR functions were substantially cheaper that the three other deployment methods. The second cheapest, competing option was

Table 5.45.: Cost (in €-Cent) Per 1,000 Requests

Endpoint	Workload	GKE-50	GKE-80	OW	GCR
Users-Signin	Linear	0,3000ct	**0,2655ct**	0,3267ct	0,3728ct
	Random	0,2632ct	0,3060ct	0,3221ct	**0,2477ct**
	Spike	0,8173ct	1,9060ct	0,6509ct	**0,2602ct**
Users-Get	Linear	0,1103ct	0,1113ct	0,5401ct	**0,0493ct**
	Random	0,1098ct	0,1101ct	0,3221ct	**0,0478ct**
	Spike	0,4425ct	0,4368ct	1,3610ct	**0,0475ct**
Devices-Add	Linear	0,1210ct	0,1291ct	0,4734ct	**0,0617ct**
	Random	0,1178ct	0,1295ct	0,5020ct	**0,0569ct**
	Spike	0,4506ct	0,4630ct	1,2797ct	**0,0586ct**
Devices-Get	Linear	0,1081ct	0,1087ct	0,3677ct	**0,0517ct**
	Random	0,1073ct	0,1079ct	0,5151ct	**0,0521ct**
	Spike	0,4496ct	0,4346ct	1,3464ct	**0,0586ct**
Sensors-Add	Sprint	0,3270ct	**0,1607ct**	19,9766ct	20,1125ct
Sensors-Get	Linear	0,1054ct	0,1053ct	0,5195ct	**0,0542ct**
	Random	0,1043ct	1,0433ct	0,4752ct	**0,0544ct**
	Spike	0,4151ct	0,4153ct	1,2230ct	**0,0586ct**
Gateway	Linear	0,1225ct	0,1276ct	0,2592ct	**0,0710ct**
	Random	0,1217ct	0,1278ct	0,4752ct	**0,0630ct**
	Spike	0,4760ct	0,4742ct	1,0206ct	**0,0539ct**
Consumers-Get	Linear	0,1157ct	0,1162ct	0,2660ct	**0,0636ct**
	Random	0,1135ct	0,1141ct	0,2636ct	**0,0624ct**
	Spike	0,4432ct	0,4439ct	1,0527ct	**0,0657ct**

Note: In each row the cheapest deployment method is highlighted in **bold**.

most of the time twice as expensive per request. This can be explained with two factors that most likely reduced the average cost. GCR serves a high amount of requests per second, while at the same time it lacks expensive fixed costs. The reservation-based fees of the GKE and OW setups are charged independently of the workload that actually occurs. One quick example illustrates this argument. In the users-get linear experiment, the GKE-50 cluster would need to serve 2.24 time more requests (405,744) at the same time with the same requests to match the costs of GCR. This phenomenon flips in the occasions, where the response time increases. Request prices rise for GCR invocations for users-signin and sensors-add. Here the fixed cost structure benefits the GKE and OW

deployments.

A surprising source of costs were the logging functionalities of the Google Cloud. Out of the 557,94€ payed in total for all experiments, 373.93€ were spent on the Cloud Logging service. Over the span of all experiment runs, a cumulatively of 845.553 GB of logs were written. The largest fraction was caused by the previously mentioned platform failure when too many sensors were added. The Kafka, Elasticsearch and Connect components tried to repair themselves and logged these attempts, while IoT core pods, that were newly created by auto-scaling in the experiment runs, tried to synchronise the mismatch between missing Kafka topics, Elasticsearch indices as well as Connect jobs and the amount of sensors saved to MariaDB. This platform behaviour triggered a cascade of log entries written to GCP's logging service. As the billing is only updated daily, the considerably increased cost were only detected on the following day.

6. Discussion

6.1. Summary

Subject of this bachelor thesis was the extensive load testing of an IoT platform to compare microservice architectures – deployed on serverful (DevOps) computing platforms – with FaaS architectures – deployed on serverless (NoOps) platforms. The following chapter will summarize the results found guided by the research questions of chapter 4.1.

The first research questions was whether there are tasks on an IoT platform that are better suited for microservices, while others might be better suited for FaaS deployments. The results have indicated no specific tasks that would clearly favour one architecture over the other. Still there were two endpoints, `Users-Signin` and `HTTP-Gateway`, on which the NoOps deployments performed better than the DevOps deployments. For the rest of the endpoints this was rather inverted, although GCR was mostly in the range of 10ms to the GKE setups. Surprisingly the differences were not as large as anticipated. OpenWhisk was able to maintain higher service levels in contrast to previous studies (e.g. [7]), but then showed similar behaviour of performance drops when reaching a threshold of approximately 2,000 requests per 10s. GCR again displayed rather robust performance results with the exception of three occasions (`Users-Signin` Random, `Consumers-Consume-Get` Linear & Random)

The second research question tried to identify differences in performance for the three request patterns – linear, random and spike workload. Could either microservice or FaaS platforms utilize one of the patterns? The GKE setups scaled in general well to demand and had very few request failures. Their performance increased synchronously to the amount of incoming requests. GCR matched this behaviour, but displayed a bit more variation in its scaling to demand. The adaptations in performance were not as evenly as for the GKE setups. The slightly higher variance on the other hand also had the positive outcome that closer to the end of experiment runs the performance quickly increased and outperformed GKE. OW had the worst performance in juxtaposition to the different workloads. At the peaks of all three workload patterns the amount of requests served dropped and response times outreached the response times of the other deployments substantially.

GKE's auto-scaling mechanisms were the subject of the third research question. GKE-50

and GKE-80 differed only slightly in their performance. From their configuration it is already predefined that GKE-50 starts new pods earlier and releases them later than GKE-80. This was also visible in the logs of the Google Cloud Platform. The effect on performance was a slight lead on GKE-80, which at the same time also resulted in more total costs per experiment run, because additional nodes were requested earlier. The cost difference disappeared when observed from a relative perspective. Costs per requests did not show any significant advantage for one of the GKE deployments. The node auto-scaling did not work as expected. Although at full scale the 40 pods were allowed to use 40 vCPUs, new nodes were rarely added. Only in a few occasions a third node was added. Four or five nodes were never used. For future studies it could be advisable to set the pods resource requests to a amount that lies even outside the hardware configuration of the node pools to guarantee that assigned resources are fully utilized.

"How does Google's container-based serverless cloud offering Cloud Run compare to the self-deployed open-source alternative OpenWhisk?" was the fourth research question guiding this thesis. The performance of OpenWhisk was for a few experiments runs surprisingly fast. This was mainly attributed to the high resource provisioning that it was granted, as node auto-scaling did not work with OW. In the majority of experiments runs however, OW could not match GCR's performance. In general OW's open-source approach provides cluster administrators and programmers with more flexibility. This flexibility comes at the price of higher complexity to get clusters set-up and configured. GCR as a fully-managed service liberates users from this burden and being KNative-based it might still provide the majority of advantages that an open-source serverless platform has.

The final research questions aimed for the pricing models and resulting cost of the different deployments. Overall GCR stood out as the cheapest of all four deployment methods. The calculations conducted did not even count into effect that idle times are not billed, while an unused GKE cluster still costs the same as a fully utilized one. The cost benefit is slightly dampened by a few performance failures observed and a larger opacity regarding the inner workings of GCR compared to the rather well-known K8-services. Overall GCR's cheap cost and good performance made it a enticing competitor to the GKE deployments.

6.2. Limitations

The most evident limitation of this bachelor thesis is the amount of tests conducted per endpoint, workload pattern and deployment method. The budget allowed only one round of experiments and it was assumed more beneficial to test more endpoints in total than testing less endpoints more thorough and replicable. While this procedure provided a

good overview over a cross-section of endpoints, failures and deviations that might only occur in rare occasions could have been interpreted in mismatch to their true proportion.

The conducted study was deliberately using K6 as the load testing tool of choice, following the choice of many practitioners as well as academics. It has served its purpose well. Common to all load testing tools though is the fact that performance evaluation is solely done on an external basis. Although metrics hereby are objective and resemble closely what the user would experience in daily-life interaction with the platform, the mechanisms behind the platform's facade are hidden. The system remains a "black box". Deviating metrics, failures or other anomalous behaviour has to be tediously matched with potential logs and are not always fully explainable. These difficulties are even more prominent in a cloud environment like the Google Cloud Platform. Especially serverless cloud services like Cloud Run abstract away a lot of the deployment details from the programmer. But by doing so, it also reduces possibilities for failure analyses as the User-Signin experiment run with a random workload has shown. The GCR deployments goodput dropped below 50% for nearly 8min. System logs did not reveal the cause of the service disruption and it can be only assumed that something failed in the auto-scaling mechanisms of GCR itself. Solving these observability and verification issues is not a trivial task as a large number of components have to work together without friction. A possible solution is introduced in the next sub-chapter that also sketches a few other potential areas of future research.

6.3. Further Research

As the limitations already stated, the IoT platform remains a black box to a load testing tool like K6. Potential future studies could address this issue and extent the explanatory power of component metrics and logs by incorporating tracing methods. Distributed tracing is an alternative monitoring approach to the more traditional metrics and logs. It is better suited to provide observability throughout the whole distributed system, because it is not limited to single system components like the former two. A trace "captures the detailed execution of causally-related activities performed by the components of a distributed system as it processes a given request" [35].

For the implementation of distributed tracing OpenTelemetry could be used. It is a projects of the Cloud Native Computing Foundation (CNCF) in the incubating stage. OpenTelemetry is „a set of APIs, SDKs, tooling and integrations that are designed for the creation and management of telemetry data such as traces, metrics, and logs“ [36].

Besides the extension through tracing and therefore a deeper understanding of performance aspects of cloud services, it could be of academic interest to deepen the knowledge

about the different cost structures in the public cloud. This thesis gave a short glimpse of how substantially cost can diverge for different deployment methods as well as IT architectures. A small exemplary calculation tried to identify a break-even point between serverful and serverless computing. This calculation assumed a constant cost function for the serverful deployment, which is rather unrealistic and favours the reservation-based cost models. In reality this cost function is more likely a staircase function as more resources have to be reserved to process an increasing workload. To find more accurate ways to calculate cost functions can be the focus of new studies, that might as well count serverless' ability to scale to zero into effect, which was excluded from investigation in this thesis.

A third and maybe most fruitful starting point of future studies could be KNative. It was briefly introduced in the chapter on the theoretical background as it is closely linked to Google's Cloud Run offering. It depicts a new major open-source serverless project that has a lot of prominent support within the industry. besides Google it is also backed by IBM, where the focus of new cloud products has shifted from OpenWhisk- to KNative-based. KNative distinguishes itself by truly preventing vendor lock-in through its ability of being deployable in numerous different ways. It would be interesting to compare how application performance and cost compare across self-hosted K8s clusters, fully managed, auto-pilot K8s clusters and direct KNative offerings. Especially the second option of open-source serverless computing on an auto-pilot cluster allowing scaling close to zero could be a worthwhile alternative to the rather complicated and costly "do-it-yourself" approach of OpenWhisk in a production environment.

The results of this thesis showed promising results of more complex applications based on NoOps cloud offerings. Google Cloud Run and underlying KNative might bridge gaps between cloud applications based on a microservice architecture and vendor lock-in prone FaaS applications through a focus on container-based deployments. DevOps still convinces by high performance and high fault-tolerance, but is more attractive – especially due to their higher costs – to applications with large amounts of users and therefore constant workloads. For smaller to medium size applications with fluctuating demand GCR and KNative represent a viable alternative with good performance and costs closer aligned to the actual workload.

List of Figures

List of Tables

Bibliography

[1] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Incorporated, 2021, ISBN: 9781492034025.

[2] Amazon Web Services. "What is AWS Lambda?" (2022), [Online]. Available: `https://docs.aws.amazon.com/lambda/latest/dg/welcome.html`. (last accessed: 15.05.2022).

[3] Google Cloud Tech. "Cloud Functions - Overview." (2022), [Online]. Available: `https://cloud.google.com/functions/docs/concepts/overview`. (published: 06.05.2022).

[4] ——, "Cloud Run Documentation." (2022), [Online]. Available: `https://cloud.google.com/run/docs#docs`. (last accessed: 15.05.2022).

[5] Microsoft. "Introduction to Azure Functions." (2021), [Online]. Available: `https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview`. (published: 28.07.2021).

[6] C.-F. Fan, A. Jindal, and M. Gerndt, "Microservices vs serverless: A performance comparison on a cloud-native web application.," in *CLOSER*, 2020, pp. 204–215.

[7] A. Jindal and M. Gerndt, "From devops to noops: Is it worth it?" In *International Conference on Cloud Computing and Services Science*, Springer, 2020, pp. 178–202.

[8] J. Lewis and M. Fowler. "Microservices – A Definition of This New Architectural Term." (2014), [Online]. Available: `https://www.martinfowler.com/articles/microservices.html`. (published: 25.03.2014).

[9] Kubernetes. "What is Kubernetes?" (2022), [Online]. Available: `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`. (published: 04.04.2022).

[10] ——, "Kubernetes Components." (2022), [Online]. Available: `https://kubernetes.io/docs/concepts/overview/components/`. (published: 30.04.2022).

[11] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," *Communications of the ACM*, vol. 64, no. 5, pp. 76–84, 2021.

[12] Apache Software Foundation. "OpenWhisk - Documentation." (2022), [Online]. Available: https://openwhisk.apache.org/documentation.html. (last accessed: 15.05.2022).

[13] Google Cloud Tech. "Google Kubernetes Engine." (2022), [Online]. Available: https://cloud.google.com/kubernetes-engine. (last accessed: 15.05.2022).

[14] ——, "Autoscaling with GKE: Clusters and nodes." (2020), [Online]. Available: https://www.youtube.com/watch?v=VNAWA6NkoBs&t=231s. (published: 07.12.2020).

[15] KNative. "KNative - Enterprise-grade Serverless on your own terms." (2022), [Online]. Available: https://knative.dev/docs/. (last accessed: 15.05.2022).

[16] Red Hat. "What is KNative?" (2019), [Online]. Available: https://www.redhat.com/en/topics/microservices/what-is-knative. (published: 08.01.2019).

[17] MariaDB. "About MariaDB Software." (2022), [Online]. Available: https://mariadb.com/kb/en/about-mariadb-software/. (last accessed: 15.05.2022).

[18] Apache Software Foundation. "Kafka - Documentation." (2022), [Online]. Available: https://kafka.apache.org/documentation/#gettingStarted. (last accessed: 15.05.2022).

[19] Elasticsearch. "What is Elasticsearch?" (2022), [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro.html. (last accessed: 15.05.2022).

[20] M. Jones, J. Bradley, and N. Sakimura, *JSON Web Token (JWT)*, RFC 7519, May 2015. DOI: 10.17487/RFC7519.

[21] Z. Jin, Y. Zhu, J. Zhu, D. Yu, C. Li, R. Chen, I. E. Akkus, and Y. Xu, "Lessons learned from migrating complex stateful applications onto serverless platforms," *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, pp. 89–96, 2021. DOI: https://doi.org/10.1145/3476886.3477510.

[22] OpenJS Foundation. "About Node.js." (2022), [Online]. Available: https://nodejs.org/en/about/. (last accessed: 15.05.2022).

[23] ——, "Express - Fast, unopinionated, minimalist web framework for Node.js." (2022), [Online]. Available: https://expressjs.com. (last accessed: 15.05.2022).

[24] Apache OpenWhisk. "OpenWhisk Actions - Creating Action Sequence." (2021), [Online]. Available: https://github.com/apache/openwhisk/blob/master/docs/actions.md#creating-action-sequences. (published: 07.10.2021).

[25] Google Cloud Tech. "Deploy a Node.js service to Cloud Run." (2022), [Online]. Available: `https://cloud.google.com/run/docs/quickstarts/build-and-deploy/deploy-nodejs-service`. (published: 12.05.2022).

[26] ——, "Understand Workflows." (2022), [Online]. Available: `https://cloud.google.com/workflows/docs/overview?hl=en`. (published: 12.05.2022).

[27] ——, "Overview of Cloud Build." (2022), [Online]. Available: `https://cloud.google.com/build/docs/overview?hl=en`. (published: 12.05.2022).

[28] ——, "Overview of Artifact Registry." (2022), [Online]. Available: `https://cloud.google.com/artifact-registry/docs/overview?hl=en`. (published: 12.05.2022).

[29] ——, "Cluster autoscaler." (2022), [Online]. Available: `https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler?hl=en`. (published: 12.05.2022).

[30] Kubernetes. "Horizontal Pod Autoscaling." (2022), [Online]. Available: `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`. (published: 30.04.2022).

[31] Helm Authors. "Helm - The package manager for Kubernetes." (2022), [Online]. Available: `https://helm.sh`. (last accessed: 15.05.2022).

[32] Apache OpenWhisk. "Scaling-up OpenWhisk Deployment on custom-built-kubernetes cluster." (2021), [Online]. Available: `https://github.com/apache/openwhisk-deploy-kube/blob/master/docs/k8s-custom-build-cluster-scaleup.md`. (published: 05.03.2021).

[33] Google Cloud Tech. "Serverless VPC Access." (2022), [Online]. Available: `https://cloud.google.com/vpc/docs/serverless-vpc-access`. (published: 12.05.2022).

[34] Grafana Labs. "What is k6?" (2022), [Online]. Available: `https://k6.io/docs/#what-is-k6`. (last accessed: 15.05.2022).

[35] Y. Shkuro, *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing, 2019, ISBN: 9781788627597.

[36] OpenTelemetry. "What is OpenTelemetry?" (2021), [Online]. Available: `https://opentelemetry.io/docs/concepts/what-is-opentelemetry/`. (published: 26.02.2021).

A. Appendix

A.1. Cost Analysis

Table A.1.: Cost (in €-Cent) Per Experiment Run

Endpoint	**Workload**	**GKE-50**	**GKE-80**	**OW**	**GCR**
Users-Signin	Linear	56,2892ct	46,0684ct	83,6619ct	88,0052ct
	Random	49,9298ct	47,7396ct	83,6619ct	92,1887ct
	Spike	43,4648ct	36,1643ct	83,6619ct	23,0677ct
Users-Get	Linear	36,1643ct	36,1643ct	83,6619ct	16,0485ct
	Random	36,1643ct	36,1643ct	83,6619ct	0,5198ct
	Spike	36,1643ct	36,1643ct	83,6619ct	3,9498ct
Devices-Add	Linear	36,1643ct	36,1643ct	83,6619ct	19,1225ct
	Random	36,1643ct	36,1643ct	83,6619ct	18,1368ct
	Spike	36,1643ct	36,1643ct	83,6619ct	4,6628ct
Devices-Get	Linear	36,1643ct	36,1643ct	83,6619ct	16,6469ct
	Random	36,1643ct	36,1643ct	83,6619ct	16,9157ct
	Spike	36,1643ct	36,1643ct	83,6619ct	4,6628ct
Sensors-Add	Sprint	7,1620ct	6,1945ct	13,9436ct	13,3547ct
Sensors-Get	Linear	36,1643ct	36,1643ct	83,6619ct	17,3000ct
	Random	36,1643ct	36,1643ct	83,6619ct	17,5051ct
	Spike	36,1643ct	36,1643ct	83,6619ct	4,6635ct
Gateway	Linear	36,1643ct	36,1643ct	83,6619ct	21,2724ct
	Random	36,1643ct	36,1643ct	83,6619ct	19,6167ct
	Spike	36,1643ct	36,1643ct	83,6619ct	4,3708ct
Consumers-Get	Linear	36,1643ct	36,1643ct	83,6619ct	19,5755ct
	Random	36,1643ct	36,1643ct	83,6619ct	19,4770ct
	Spike	36,1643ct	36,1643ct	83,6619ct	5,0899ct