# Emulate typical edge/cloud applications

G. Stekas
g.stekas@student.tue.nl, ID: 1532219

***Abstract*—This report delves into the proceedings of a Bachelor End Project named Emulate typical edge/cloud applications. The project investigates the performance of applications in Kubernetes, under varying workloads. An Apache Kafka - Apache Flink data pipeline as well as a microservice based hotel reservation application are deployed and user behavior is simulated. The implementations will then be publicly available to facilitate future research on performance and scalability studies.**

## I. Introduction

In today's digital world, web applications play a significant role in the interactions of people. Applications are to be found anywhere, between businesses and consumers, in e-commerce, in social media, in multimedia streaming or in e-mail, areas integral to our daily lives. Hence, meeting the performance requirements of these applications is a necessity, especially when it comes to time-sensitive, critical systems (like financial systems, emergency response systems, defensive systems etc.) [1]. It is crucial that these applications run at lightning speed to keep up with the evolving demands of the modern world, as any performance degradation can lead to significant impacts on user experience. In the case of critical systems, performance degradation could prove to be catastrophic for business operations.

This is where containerized environments, like Kubernetes, come into play. Containerized environments are used in production, as well as for simulation purposes. Through workload simulation, the behavior of applications under various conditions can be explored, together with their scalability and reliability. Potential bottlenecks on the deployment of these applications can be identified and thus, the quality of the services provided can be conserved on a high level, ensuring a user-centric approach.

As far as novelty goes, Kubernetes is somewhat novel, having being released less than 10 years ago, but this does not mean that there are not any existing contributions. The Kubernetes Scheduler and auto-scheduling techniques have already been explored [2], [3] along with workload generation [4]. Application performance has also been explored, but it mostly consists of implementations not publicly available [5], [6]. Open-source projects can only be found for microservice applications, such as the DeathStarBench benchmarking suite [7], which also includes applications whose performance has not been analyzed and thus will be used in this project. This paper aims to explore the performance of big data as well as microservices applications, investigate how to easily test different solutions to simulate real deployments and set the foundation for future research while providing the project implementations to be publicly available.

## II. Methods

In order to deploy and emulate the behavior of the selected applications, a setup of three virtual machines is used. Each virtual machine consists of a 4-core CPU, 16 GB RAM memory and a storage disk of 100 GB. These three form a Kubernetes cluster. Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation [8]. Kubernetes is built to be used anywhere, allowing applications to be run across on-site deployments and public clouds, as well as hybrid deployments in between [9]. A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node. These worker nodes host the pods that are the components of the application workload. The control plane manages the worker nodes and the pods in the cluster [8]. In essence, the containerized applications are running inside the pods.

As the aim of the project is to study applications of different types and investigate how these perform under varying workload, two different applications were chosen. The first one is a big-data application, where real-time events are processed, while on the other hand, a microservice application where asynchronous processing takes place is studied.

### *A. Apache Kafka - Apache Flink Data Pipeline for Click Fraud Detection*

The first chosen application consists of an Apache Kafka and Apache Flink data pipeline. Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications. Event streaming is the practice of capturing data in real-time from event sources like databases for later retrieval, manipulating, processing, and reacting to the event streams in real-time as well as retrospectively [10]. Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams [11]. The Flink runtime consists of two types of processes: a JobManager and one or more TaskManagers. In Flink, the JobManager coordinates the execution of the Job and the TaskManagers perform the actual work. TaskManagers contain slots, where the tasks of the Job are executed.

In essence, a Kafka cluster server and a Flink instance are deployed in the Kubernetes cluster and a click fraud detection application is deployed as a Flink job [12]. Click fraud is a type of fraud that occurs on the Internet in pay-per-click (PPC) online advertising [13]. The Flink job receives records (in the

form of produced messages) from a Kafka topic in the cluster, which have the following format:

```
{
    "ip": "181.1.44.256",
    "userID": "d58c9d8e-2091-001f-1b88-14
       g5a632e218",
    "timestamp": 1595431611,
    "eventType": "display/click"
}
```

These messages are produced by a Python script, which generates randomized records of this format and features a set sleep time (i.e. the production rate of the messages can be adjusted). This production rate signifies the workload for this application. The Flink job searches for specific patterns in the records, in order to detect fraud and then has the ability to publish the results in a new Kafka topic. Figure 1 presents a visual representation of the pipeline.

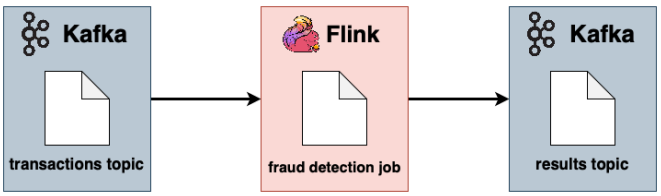


Fig. 1. Visual representation of the Flink-Kafka data pipeline

In order to uncover relations and reach meaningful deductions between the workload and the application's response, Prometheus alongside Kafka Exporter are used. Prometheus is an open-source systems monitoring and alerting toolkit, with its main component being the Prometheus server, which scrapes and stores time series data [14]. Kafka Exporter exposes the metrics of the Kafka cluster to Prometheus. A high-level workflow diagram is provided in Figure 2.

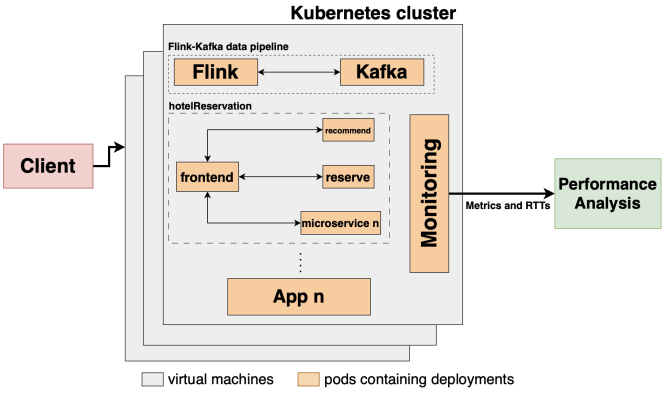


Fig. 2. High-level diagram

The metrics `kafka_consumergroup_lag` and `kafka_topic_current_offset` are selected. The `kafka_consumergroup_lag` represents the lag of the Kafka consumer, so effectively how many records is the pipeline "behind" in processing from the generated messages or in other words how large is the queue in the pipeline. `kafka_topic_current_offset` is a pointer to the latest message produced in the topic. By calculating the rate of change of this metric, the records per second generated are obtained. For conducting an experiment and gathering results, a script that generates records to the Kafka topic is written in Python. More specifically, the sleep time starts at half a second, so effectively two messages are produced each second. This sleeping time is halved every 30 seconds, so the rate of records per second is doubled.

### *B. Hotel Reservation application consisting of microservices*

The second chosen application is the `hotelReservation` application from the open-source benchmark suite DeathStarBench [7]. Microservices are applications structured as a collection of services that are independently deployable and loosely coupled [15]. In that sense, `hotelReservation` consists of 11 microservices that can be independently deployed in the Kubernetes environment. This application represents a hotel booking reservation system, which includes recommendations, user reviews and the functionality to place a hotel reservation. In the open-source repository, a workload script, written in Lua, that comprises of different HTTP Requests, like POST and GET is included. The aforementioned script simulates common user behavior on a hotel booking platform. These requests are generated to the frontend of the application using wrk2, a tool based on wrk, using its' Lua API, with an improved latency measuring technique [16]. Wrk is a modern HTTP benchmarking tool capable of generating significant load when run on a single multi-core CPU [17]. In more detail, wrk2 (just as wrk) provides a command line to adapt the duration and the requests per second of each workload test. It replaces wrk's individual request sample buffers with HdrHistograms. This allows it to record the full latency details without loss [16].

Latency is the amount of time it takes from when a request is made by the user to the time it takes for the response to get back to that user [18]. The latency that wrk2 calculates, represents the time taken for a request to be processed. It is the duration from when a request was supposed to be sent (according to the constant request rate plan) to when the response is received in the frontend [16]. In more detail, it includes:

- **Queuing Time**: The time the request waits in a queue until it can be sent over the network.
- **Network Latency**: The time it takes for the request to travel from the client to the server and back.
- **Server Processing Time**: The time the server takes to process the request and generate a response.

Moreover, it provides statistics on how many requests were successfully processed on the target address (frontend). This is used to validate the impact on the performance of the system under varying workload tests.

It needs to be noted, that the latest version of the application available in GitHub, has errors in the deployment .YAML files for Kubernetes, not allowing for the app to be deployed. These were successfully fixed in the course of this project. In order to

conduct workload experiments, a fixed duration of 20 seconds and an increasing, by a step of 1, requests per second (RPS) variable are used. The high-level principle shown in Figure 2 is also followed here.

## III. Results and Discussion

### A. Apache Kafka - Apache Flink Data Pipeline for Click Fraud Detection

Below, the results of the experiment on the pipeline are presented:

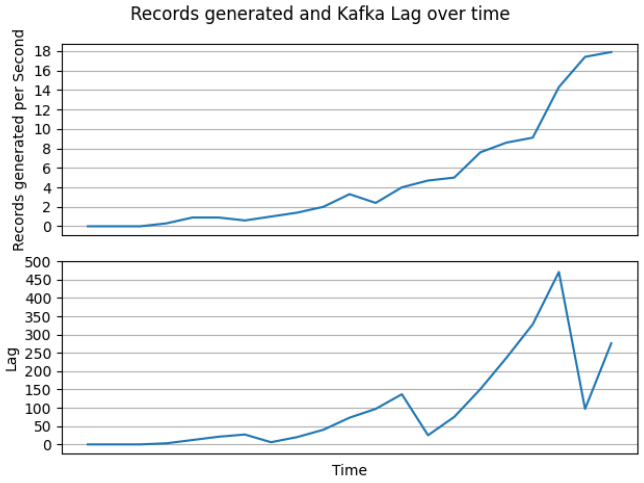


Fig. 3. Results for an experiment with a linearly increasing production rate

In Figure 3, the records generated, so the increasing workload, together with the average lag are plotted in the y-axis of each sub-plot, over the same time period, in the x-axis.
As it can be observed, the lag increases proportionally with the increase of the rate of the records generated. A noticeable spike occurs after the rate reaches and surpass the value of 5 records per second with the lag surpassing 450, effectively meaning that 450 records are waiting to be processed in the queue.
As this pipeline serves a real-time, time critical application, such an increase in the lag could be problematic and disastrous.
However, it needs to be stated that the performance of the pipeline is heavily dependent on the system's resources, so the processing power, as well as the storage's capacity and data transfer speed. If more resources were to be allocated, then an improved result can be obtained.
On the same note, allocating more task slots would mean that the Flink job would run multiple times in parallel and more records would be processed at the same time. Moreover, increasing the parallelism level on Flink which determines how many parallel instances (subtasks) of an operator are created could prove to be helpful. These points can be used in further research and analysis.

### B. Hotel Reservation application consisting of microservices

For the workload experiment on `hotelReservation`, the following results were obtained:

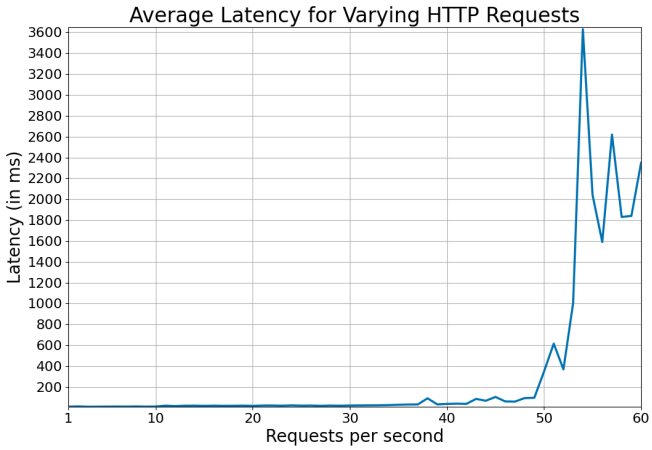


Fig. 4. Average latency results for increasing RPS values

Figure 4 depicts the average latency in milliseconds for an experiment with linearly increasing HTTP Requests per second rate.
It can be noticed that up until a RPS rate of 40, the latency gradually increases but it is mostly low, between 20 $ms$ and less than 100 $ms$. A noticeable increase takes place when the RPS rate reaches 50, where a spike reaching an average latency value of more than 3.6 seconds occurs. This is quite a significant value and signifies a significant performance degradation that affects user experience.

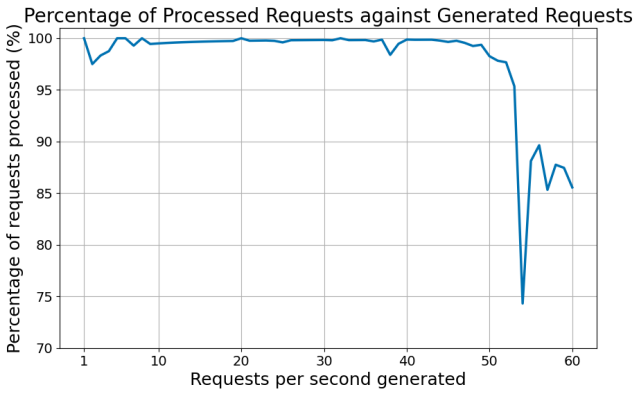


Fig. 5. Percentage of processes requests against the generated requests of the linearly increasing workload experiment

This plot serves to visualize the percentage of actual processed requests in the frontend for each RPS value of the experiment. As it can also be observed in Figure 5, when the RPS rate reaches 50, the performance of the setup significantly decreases and after 50 RPS, less than 75% of the requests are processed in the frontend. It needs to be noted that beyond this point, the performance keeps decreasing significantly until the pods crash. Nevertheless, an average user will complete around 10 to 15 requests while completing a hotel reservation in total. This means that the system under discussion can handle up to around 3 to 4 complete reservation procedures each

second. When considering the actual duration needed for an average user to complete such a procedure, this performance is substantial.
Once again, as with the Flink-Kafka pipeline, the performance is resource dependent. A system with more resources would most likely be bound to better results, especially considering the fact that after performance degradation is faced, the pods of the deployment ”die”.
It should also be noted that an experiment with the deployment scaled by 2 (so double instances created, meaning two pods running each microservice) was also carried out, but the results obtained did not differentiate enough to be included and taken into account in this paper. This could account to the application not consisting of a proper LoadBalancer, but due to the limited time frame, this was not further explored. However, this can looked into further in future research, with different system properties to experiment with.

## IV. Conclusions

Workload experiments on two different applications have been conducted. Their performance has been explored, with results analyzed and points for future improvement and studies drawn. The implementations have become open-source.[1]
For the Kafka-Flink pipeline, a representative real-time, big data implementation has been studied, with the performance analyzed. Results follow a linear trend between the dependent and the independent variables, with the performance threshold being around 5 records generated (in the queue) per second. Points such as increasing the parallelism and task slots in Apache Flink have been drawn for future exploration.
On the same note, for the `hotelReservation` application, typical user behavior was succesfully emulated, even representing significant workload. Performance was analyzed and results provided an indication of the of the system. Moreover, issues present on the current version of the application were fixed, so it could be deployable.
A general conclusion that can be made for both applications is that their performance increases with the increased workload. The system in test can keep up with an adequate load for both application types.
The `hotelReservation` application reaches a higher absolute value of RPS, but this can be credited to HTTP requests being less resource demanding than big data processing in the Flink-Kafka pipeline, as well as to the fact that the latter one involves synchronous, real-time data generation.
Furthermore, it is of high significance to note that the main limitation are system resources and a fact that needs to be considered on future contributions. Finally, a solid foundation has been established for further testing on these applications with ease, taking advantage of Kubernetes' features, such as deployment scalability.

[1] https://github.com/stekas7/BEP-Emulate-typical-edge-cloud-applications

## V. Learning curve

In this Bachelor End Project, a very steep learning curve has been encountered. At the start of the project, there was no experience with distributed systems and containerization in general present, also with Linux, which is the operating system of the virtual machines used for the cluster. Since then, the basics of distributed systems and Kubernetes have been explored, and experience on working confidently on a Kubernetes cluster has been established.
Moreover, knowledge on the field of edge and cloud applications has been acquired. Furthermore, acquaintance with the basics of Apache Flink and Apache Kafka, as well as understanding of the essentials of HTTP requests and benchmarking have been attained. Finally, methods of extracting and analysing metrics, such as Prometheus and the meanings of latency have been explored.

## References


[1] “What is web performance?” 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Performance/What_is_web_performance
[2] T. Watts, D. Bourrie, R. Benton, and J. Shropshire, “Insight from a containerized kubernetes workload introspection,” *Proceedings of the 54th Hawaii International Conference on System Sciences — 2021*, 2021.
[3] K. Senjab, S. Abbas, N. Ahmed, and et al., “A survey of kubernetes scheduling algorithms,” *Journal of Cloud Computing*, 2023.
[4] B. Chiminelli, “Optimizing resource allocation in kubernetes: A hybrid auto-scaling approach,” 2023.
[5] C. Zhu, B. Han, and Y. Zhao, “A comparative performance study of spark on kubernetes,” *The Journal of Supercomputing*, 2022.
[6] C. Cardas, J. Aldana-Martín, A. Burgueño-Romero, and et al, “On the performance of sql scalable systems on kubernetes: a comparative study,” 2022.
[7] Y.Gan and et al, “An open-source benchmark suite for microservices and their hardware-software implications for cloud edge systems,” 2019.
[8] “Kubernetes documentation,” 2023. [Online]. Available: https://kubernetes.io/docs/concepts/overview/
[9] “What is kubernetes?” 2024. [Online]. Available: https://cloud.google.com/learn/what-is-kubernetes
[10] “Apache kafka documentation,” 2023. [Online]. Available: https://kafka.apache.org/documentation/#gettingStarted
[11] “Apache flink documentation,” 2024. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-stable/
[12] “Click fraud detection flink job,” 2020. [Online]. Available: https://github.com/Nada-S/Clicks_fraud_detection_with_Kafka_and_Flink
[13] “Click fraud,” 2024. [Online]. Available: https://en.wikipedia.org/wiki/Click_fraud
[14] “Prometheus documentation,” 2024. [Online]. Available: https://prometheus.io/docs/introduction/overview/
[15] “What are microservices?” 2024. [Online]. Available: https://microservices.io
[16] “giltene/wrk2: A constant throughput, correct latency recording variant of wrk,” 2019. [Online]. Available: https://github.com/giltene/wrk2#
[17] “wg/wrk: Modern http benchmarking tool,” 2021. [Online]. Available: https://github.com/wg/wrk
[18] “Understandind latency,” 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Performance/Understanding_latency#