Seamless Migration of Containerized Stateful Applications in Orchestrated Edge Systems

Roman Kudravcev[1,*], Sebastian Böhm[1]

[1]*University of Bamberg, An der Weberei 5, Bamberg, 96047, Germany*

Abstract

Edge and fog computing has established an innovative approach in the distributed systems context and enhanced traditional cloud computing by improving latency, bandwidth utilization, and data protection. To orchestrate such environments, dynamic changes in load, like the number of edge devices, must be considered and often result in the need to migrate services to other nodes. This also requires the seamless migration of highly available stateful services to handle such changes. In this paper, we propose a tool for seamless service migration while addressing critical issues like state management, networking, and service availability. We perform a real-world experiment and quantitatively evaluate resource utilization, response time, and availability during the migration and idle states of the involved Kubernetes clusters. We show that it is possible to provide a tool that almost achieves a seamless migration experience with high availability to enable changes for orchestrated edge environments.

Keywords

Edge Computing, Kubernetes, Service Migration, Workload Migration, Orchestration

1. Introduction

Edge and fog computing have emerged as innovative distributed computing paradigms, extending traditional cloud infrastructure capabilities. They aim to meet the growing demand for real-time, latency-sensitive applications and data processing closer to the source, such as Internet of Things (IoT) devices [1]. These paradigms significantly enhance traditional cloud infrastructure by improving latency, bandwidth utilization, and data protection, enabling efficient real-time applications [2]. Cloud-edge orchestration is essential for managing workloads effectively across cloud, edge, and IoT layers. Ensuring efficient application placement, reducing latency, and optimizing resource usage is critical. By dynamically assigning workloads and making policy-driven decisions, orchestrators enable scalability, fault tolerance, and seamless operation, even in complex, distributed environments [3]. Despite its importance, cloud-edge orchestration faces several challenges, including service placement. Service placement is a key challenge in edge and fog computing. It involves determining the optimal deployment of services or applications within a distributed network to minimize latency, energy consumption, and bandwidth usage while maximizing resource availability and ensuring network reliability [4]. However, dynamic changes in load, the number of edge devices, and their locations often require services to be migrated to alternative nodes to maintain performance [5]. This migration is especially challenging for certain applications that cannot tolerate downtime. For example, applications that require continuous availability or state persistence during migration (stateful applications).

Currently, stateful migration and request forwarding are not implemented or evaluated during the migration process, although other studies have considered service placement and migration of stateless applications. Also, there is no comprehensive literature on this particular problem. Hence, this paper's objective and contribution are designing, implementing, and validating a tool for seamless application migration in edge and fog computing environments. To address this gap, the proposed tool integrates self-developed methods and existing technologies to address critical issues such as state management,

Vienna'25: 17th Central European Workshop on Services and their Composition (ZEUS), February 20–21, 2025, Vienna, Austria

*Corresponding author.

roman.kudravcev@stud.uni-bamberg.de (R. Kudravcev); sebastian.boehm@uni-bamberg.de (S. Böhm)
https://github.com/romankudravcev (R. Kudravcev); https://www.uni-bamberg.de/en/pi/team/boehm-sebastian/ (S. Böhm)
0000-0003-3719-1923 (S. Böhm)

networking, and service availability. A key aspect of this research is benchmarking and experimental evaluation. This is done to assess the performance and effectiveness of the tool during migration. The benchmarking process measures system availability, resource utilization, and migration time. This comprehensively analyzes the tool's impact before and during migration.

We take a Design Science Research (DSR) approach. We focus on developing and evaluating a migration tool for edge orchestration environments. The tool is implemented to address key challenges such as state management and availability. Its effectiveness is validated through quantitative benchmarking. Metrics such as availability, response time, resource utilization, and migration duration are measured to assess its efficiency and impact on system performance.

The rest of the paper is organized as follows: First, Section 2 discusses existing approaches to migration in cloud- edge orchestration and the current state of research. After that, we cover the concepts of the tools used for the migration tool (Section 3), which helps to understand how the individual components work. Then, we examine the implementation of the migration tool and investigate the individual problems that need to be solved for a successful migration (Section 4). Section 4 will also look at these problems and explain which tools are used to solve them and how they are used. We will then evaluate the migration tool, looking at predefined metrics and comparing them to the system in an idle state (Section 5). Finally, we review the tool and the evaluation (Section 6) and outline plans for further experimental studies (Section 7).

2. Related Work

Ma et al. [6] propose a framework for efficient live migration of edge services using Docker containers, using their layered storage architecture to minimize the amount of data transferred during the migration. However, the work focuses on single-container migrations. It does not address orchestrated multi-container setups or stateful applications that require state management.

Similarly, Kaur et al. [7] investigate live migration of containerized microservices across Kubernetes (K8s) clusters. Their solution ensures uninterrupted communication with the migrated services using Traefik ingress controllers and DNS redirection. While effective, this approach relies on manual configuration and primarily targets stateless workloads. It leaves gaps in automating migration for stateful applications and handling dynamic resource management in orchestrated environments.

This research addresses these limitations with an automated tool for stateful and stateless migration in orchestrated edge systems. The proposed solution focuses on seamless state management and real-time request forwarding to ensure minimal downtime and consistent performance during migrations.

3. Background

3.1. Migration

Migration generally encompasses moving data, applications, or systems from one environment to another. Typical scenarios include moving workloads between cloud platforms, data centers, or storage systems. In edge computing, workload migration is used to offload workloads from cloud data centers to edge nodes to improve latency and bandwidth for end users. Particularly stateful and stateless application migration are in focus. Different strategies for workload migration are available, with benefits and downsides regarding migration time and performance degradation, as shown by [8].

Each strategy requires careful planning to address compatibility, security, and minimizing downtime. This holds explicitly when multi-tier applications, consisting of Database (DB)s and caches, must be moved with minimal downtime.

The previously mentioned stateful application stacks store information about past interactions, such as session data or DB records. To ensure continuity and proper functionality, migrating stateful application stacks involves transferring both the application and its state. This includes maintaining service availability while migrating DBs and session data [9].

On the other hand, stateless applications retain no information about previous interactions and treat each request independently. Migrating stateless applications is more manageable because only the application itself needs to be transferred, with no state synchronization or data migration required [10].

3.2. Tunneling

Tunneling is a networking technique that allows secure data to travel over public networks intended for private use. It creates a direct connection between two networks by encapsulating data packets, allowing them to traverse networks that do not natively support the original protocol. This encapsulation also supports encrypted communications to ensure data security.

Tunneling is commonly used in Virtual Private Networks (VPNs) to bypass firewalls, support unsupported protocols, and establish secure connections. It simplifies communication between networks without requiring extensive configuration or routing through multiple servers [11, 12].

However, tunneling has its drawbacks. Encapsulation consumes resources, which can slow down communication. In addition, while packets are encrypted, tunnels bypass firewalls. This poses a security risk if unauthorized access is gained. Proper management is essential to mitigate these vulnerabilities [13, 14].

4. Implementation

4.1. Networking

A secure connection is inevitable for the migration process. Both environments must communicate to replicate the DB to the target environment. Since the DBs may contain sensitive data, unauthorized external access must be prohibited. Therefore, we want to use a tunneling tool to connect container orchestration platform clusters. We solved this problem with Submariner[1]. Submariner is an open-source project that enables seamless networking between Pods and Services across multiple K8s clusters, regardless of whether running on-premises or in the cloud. It provides cross-cluster Layer 3 (L3) connectivity using encrypted or unencrypted connections. This is realized with an tunnel using Virtual eXtensible LAN (VXLAN). This allows workloads in different clusters to communicate as if they were on the same network. Submariner is designed to be network plug-in (CNI) agnostic to ensure compatibility with various K8s networking setups.

After migration, handling requests sent to the source environment is critical, often because IoT devices have not yet been updated to point to the target environment. We assume that devices that receive a response from the origin environment will also receive information about the target for subsequent requests, as similarly discussed in [15]. To ensure uninterrupted service, we implemented a mechanism to forward requests from the origin to the target environment, assuming the target has the current DB state and primary role. The solution uses an HTTP reverse proxy deployed in the source environment. This proxy intercepts HTTP requests, replicates their details (method, headers, body), and forwards them to the target environment. It then returns the target response to the client, preserving headers and status codes for transparency. Although designed for HTTP, this approach can be adapted for TCP or UDP traffic. The proxy forwards raw byte streams for TCP, and for UDP, it forwards datagrams. This flexible forwarding mechanism ensures seamless communication across protocols.

4.2. Migration

To enable the migration of our system, we developed a tool written in the programming language Go. This tool scans the resources within a K8s cluster, such as deployments, services, ingress routes, config maps, and secrets. It then checks whether these resources already exist in the target environment. If

[1]https://submariner.io/

they are not, the tool cleans up the resources by removing unique identifiers (e.g., creation dates and IDs) before applying them to the new cluster.

For data migration, we focus on SQL DBs, specifically PostgreSQL, which is the most popular SQL DB.[2] In container orchestration platforms like K8s, SQL DBs are typically deployed using stateful sets or operator-managed DB clusters. Operator-managed clusters handle tasks such as high availability, scaling, rolling updates, resource management, and security, making them a robust choice for database management.[3]

We used the CloudNativePG[3] operator for testing. Many PostgreSQL operators, including CloudNativePG, support bootstrapping a new DB from an existing one. During migration, the target DB cluster is bootstrapped as a replica of the source DB, which continues to run as the primary DB. Once the target environment is synchronized with the source, applications, and data are fully migrated. At this point, the roles of the clusters must be switched: the replica is promoted to primary, and the original primary is demoted to the replica. This ensures the target environment can handle writes without errors since replicas typically do not allow writes. In CloudNativePG, this process includes synchronized demotion and promotion of DB clusters.[3] The process is similar for PostgreSQL, deployed in a StatefulSet, but requires additional manual steps. First, we enable logical replication on the source DB so the target can replicate it. Next, we enable the publishing of changes on the source. The DB schema is then imported from the source, and the target DB subscribes to the source's publication. Once replication is complete, we can switch to the target DB by disabling the subscription.

5. Evaluation and Results

5.1. Experimental Setup and Design

To evaluate the migration's impact on the environments and key metrics, we designed a controlled experimental setup to ensure that the results were reproducible and consistent.[4] Therefore, we used two Ubuntu 20.04 Virtual Machines (VMs) with 2 vCPUs, 4 GB memory and an SSD with a capacity of 30 GB each. Both VMs run on-premises on one physical host machine with Kernel–based Virtual Machine (KVM) as hypervisor and containerd as container runtime. These VMs were split into two single-node clusters: a origin cluster for migration and a target cluster as the destination.

The setup consists of three components: a resource utilization collector, a test application, and a client application. The resource utilization collector used K8s' Metrics API[5] to monitor cluster-wide CPU and memory usage each second, storing the data with timestamps in an SQLite DB. The test application is a message store with a REST API for storing, retrieving, and deleting messages. It uses a PostgreSQL DB managed by the CNPG operator. A message contains a time stamp, a unique ID, and a message so we can later check which message may not have been received. The client application is a lightweight HTTP client that simulates an IoT device and is configured to send requests to the test application's REST API. The client supports adjustable request rates and GET/POST ratios. It logs the details of each request, namely HTTP method, message content, success status, timestamp, and response time. After all requests have been sent, the client retrieves the stored messages from the test application to determine message loss, availability, and average response time. A GET request is considered as failed if the client receives an HTTP status code outside the 200 range. The number of failed POST requests is determined by looking if the DB contained the unique message sent by the client.

Our evaluation consists of two scenarios. The first scenario measures the idle load without migration to establish a baseline for performance and resource utilization under normal conditions. The second scenario evaluates the load during migration. By testing these two scenarios, we can compare performance and resource utilization between the two scenarios. We can also observe and evaluate

[2] https://survey.stackoverflow.co/2024/technology#1-databases
[3] https://cloudnative-pg.io/
[4] Tool and resources for the performed experiment available online: https://github.com/romankudravcev/clustershift-benchmark
[5] https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/

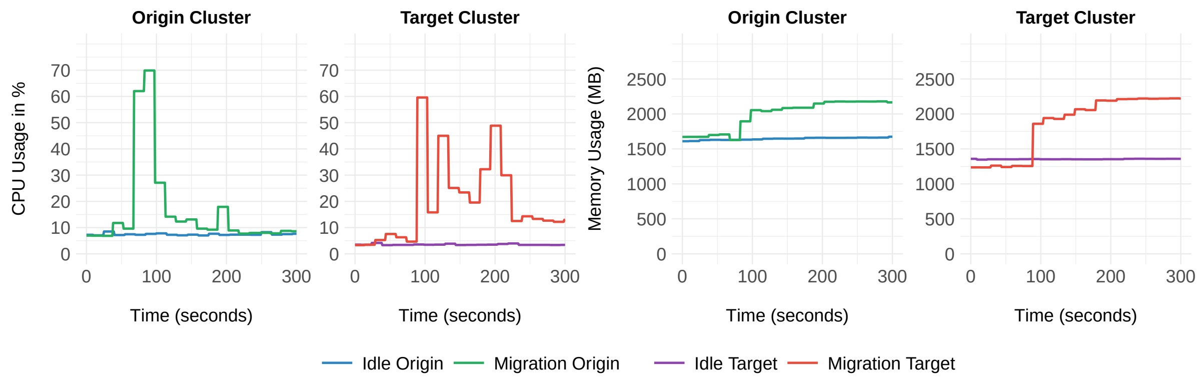

Figure 1: CPU and Memory Utilization of the Origin (left) and Target (right) K8s Clusters.

availability and downtime during a migration. We deploy the resource utilization collector and the test application on that cluster and start making requests with our implemented client. For testing during migration, we deploy the resource utilization collector on both clusters to get an overview of the resource utilization of both clusters and deploy the test application on the origin environment. Then, we start making requests with our implemented client and start our migration process.

5.2. Experimental Results

Figure 1 shows the CPU and memory utilization of the origin and target clusters during the idle and migration states. Idle corresponds to normal cluster load with no migration components, while migration includes deploying the migration components and executing the migration process. In the origin cluster, at about 30 seconds, a small spike in memory usage can be observed. This reflects the deployment of the Submariner Broker and Operator, which requires about 30 MB additional memory. A similar increase can be observed in the target cluster at roughly 40 seconds. The CPU utilization also experiences an increase due to the deployment of the Submariner resources. It increases from approximately 6.85% to a peak of 11.75% on the source cluster and from 3.45% to a peak of 7.55% on the target cluster. At 100 seconds the most significant jump in both CPU and memory utilization is noticed, caused by the replication of the PostgreSQL DB. A memory spike of 430 MB on the origin cluster and 605 MB on the target cluster is observed. Both clusters experience a peak memory load of approximately 2200 MB, which remains until the end of the experiment. On the CPU side, the origin cluster peaks at 70% utilization, while the target cluster peaks at 60%. Both peaks are also caused by the replication of the DB. After the completion of the migration process, the replica DB is promoted to primary, and the original primary is demoted and decoupled from the target DB. This results in a drop in the CPU utilization to about 8% on the origin cluster and 13% on the target cluster.

Figure 2 shows the response time of our deployed test application comparing idle and migration states for GET and POST requests. In the idle state, the average response time for GET requests is 34.5 ms ($\sigma = 2.83$). While migration, the average response time for a GET request increases to 35.9 ms ($\sigma = 6.16$). For POST requests, the average response time during idle is 133.47ms ($\sigma = 69.76$), while during migration, it decreases slightly to 118.17ms ($\sigma = 74.23$). Notably, the migration process causes a considerable number of outliers during migration for GET requests.

To evaluate availability and downtime during the migration we check the number of failed requests and their timestamps. A total of 2165 requests were sent, with approximately 70% being POST requests (1515) and 30% (650) being GET requests. Out of 650 GET requests, 12 failed (1.85%). For POST requests, 29 out of 1515 failed (1.91%). Overall 41 out of the 2165 sent requests failed, resulting in an availability of 98, 11%. The total downtime during the migration was 4.66 seconds. This was calculated by looking at the timestamps of failed requests.

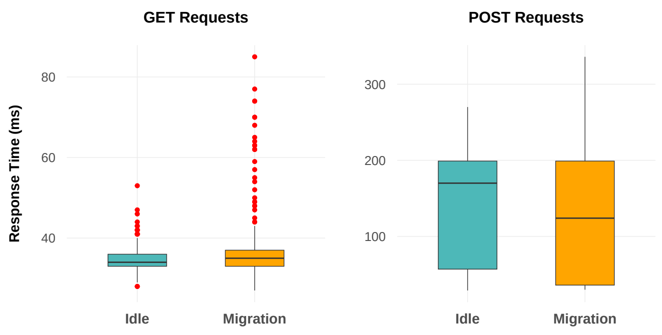

Figure 2: Response Time of the Test Application during Idle and Migration States for GET and POST Requests.

6. Discussion

The results show that Submariner has minimal impact on CPU and memory utilization, supporting its use in migration scenarios. DB replication, on the other hand, caused more significant spikes in resource usage. This is expected because replication requires additional resources to transfer large amounts of data and ensure data consistency. However, it is important to consider clusters with higher idle loads, as the replication process could potentially overload such clusters. Response time differences are minor for this use case and can be considered unimportant. It is relevant when clients don't automatically update their connections based on the response received from the application. In this case clients continue routing through the reverse proxy until the target cluster's IP is manually configured. This results in an overhead since we forward the request not just once per client, but until the IP switch to the target cluster is completed. This configuration directly affects the overall response latency. An availability rate above 98% confirms that the tool is suitable for a migration use case. The duration of the downtime shouldn't be influenced by the size of the DB. The application startup times will most likely affect the downtime because, at boot up, because the target DB must be ready for write operations.

The proposed experiment and the obtained results underlie a few limitations. Firstly, not all types of data were considered during the migration. In particular, no user context (i.e., state of the test application) or session data, such as a Redis store, was included in the test application. The evaluation was also limited to an SQL DB managed by an operator. This only covers a small portion of use cases. This is a proof of concept and we are only showing general applicability. Finally, we also have threats to validity in our experimental setup, for example application startup and network stability, which could impact the results.

7. Conclusion and Future Work

This paper showed an implementation of how a tool for migrating orchestrated environments could be built and an evaluation of this tool. To highlight the contribution of our tool, we conclude that we achieved an almost seamless application migration in edge and fog computing environments with an availability of over 98%. With our tool we are able to establish a secure tunnel between our environments by using Submariner, migrate stateful application by replicating PostgreSQL DBs and forward incoming traffic by rerouting it with a reverse proxy.

While our implementation and evaluation of this tool provided important insights, it also highlighted issues that require further investigation. The evaluation of this tool was performed using single node clusters, which does not reflect reality. In future experiments, it would be interesting to benchmark multi-node clusters to see if the tool's results are comparable. In order to handle traffic forwarding in a more efficient and generic way, different proxy solutions or service meshes that also sound promising, should be investigated in future work. It would also be interesting to look at migrating different DB types such as NoSQL or key-value stores to cover a wider variety of storage methods.

Declaration on Generative AI

During the preparation of this work, the authors used DeepL and ChatGPT-4 in order to: Grammar, spell check, and rephrasing. After using these tools/services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] K. Cao, Y. Liu, G. Meng, Q. Sun, An Overview on Edge Computing Research, IEEE Access 8 (2020) 85714–85728. doi:10.1109/ACCESS.2020.2991734.

[2] F. A. Salaht, F. Desprez, A. Lebre, An Overview of Service Placement Problem in Fog and Edge Computing, ACM Computing Surveys 53 (2021) 1–35. URL: https://dl.acm.org/doi/10.1145/3391196. doi:10.1145/3391196.

[3] S. Böhm, G. Wirtz, Cloud-Edge Orchestration for Smart Cities: A Review of Kubernetes-based Orchestration Architectures, EAI Endorsed Transactions on Smart Cities 6 (2022) e2. doi:10.4108/eetsc.v6i18.1197.

[4] R. Zheng, J. Xu, X. Wang, M. Liu, J. Zhu, Service placement strategies in mobile edge computing based on an improved genetic algorithm, Pervasive and Mobile Computing 105 (2024) 101986. doi:10.1016/j.pmcj.2024.101986.

[5] C.-H. Hong, B. Varghese, Resource Management in Fog/Edge Computing: A Survey on Architectures, Infrastructure, and Algorithms, ACM Computing Surveys 52 (2020) 1–37. doi:10.1145/3326066.

[6] L. Ma, S. Yi, N. Carter, Q. Li, Efficient Live Migration of Edge Services Leveraging Container Layered Storage, IEEE Transactions on Mobile Computing 18 (2019) 2020–2033. URL: https://ieeexplore.ieee.org/document/8470949/. doi:10.1109/TMC.2018.2871842.

[7] K. Kaur, F. Guillemin, F. Sailhan, Live migration of containerized microservices between remote Kubernetes Clusters, in: IEEE INFOCOM 2023 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), IEEE, Hoboken, NJ, USA, 2023, pp. 1–6. URL: https://ieeexplore.ieee.org/document/10225858/. doi:10.1109/INFOCOMWKSHPS57453.2023.10225858.

[8] J. Zheng, T. S. E. Ng, K. Sripanidkulchai, Workload-aware live storage migration for clouds, in: Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, ACM, Newport Beach California USA, 2011, pp. 133–144. doi:10.1145/1952682.1952700.

[9] S. Wang, J. Xu, N. Zhang, Y. Liu, A Survey on Service Migration in Mobile Edge Computing, IEEE Access 6 (2018) 23511–23528. doi:10.1109/ACCESS.2018.2828102.

[10] F. Barbarulo, C. Puliafito, A. Virdis, E. Mingozzi, Extending ETSI MEC Towards Stateful Application Relocation Based on Container Migration, in: 2022 IEEE 23rd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM), IEEE, Belfast, United Kingdom, 2022, pp. 367–376. doi:10.1109/WoWMoM54355.2022.00035.

[11] Z. Aqun, Y. Yuan, J. Yi, G. Guanqun, Research on tunneling techniques in virtual private networks, in: WCC 2000 - ICCT 2000. 2000 International Conference on Communication Technology Proceedings (Cat. No.00EX420), volume 1, 2000, pp. 691–697 vol.1. doi:10.1109/ICCT.2000.889294.

[12] IP in IP Tunneling, RFC 1853, 1995. URL: https://www.rfc-editor.org/info/rfc1853. doi:10.17487/RFC1853.

[13] J. Hoagland, S. Krishnan, D. Thaler, Security Concerns with IP Tunneling, RFC 6169, 2011. URL: https://www.rfc-editor.org/info/rfc6169. doi:10.17487/RFC6169.

[14] T. Saad, B. Alawieh, H. T. Mouftah, S. Gulder, Tunneling techniques for end-to-end vpns: generic deployment in an optical testbed environment, IEEE Communications Magazine 44 (2006) 124–132.

[15] U. Bulkan, T. Dagiuklas, M. Iqbal, K. M. S. Huq, A. Al-Dulaimi, J. Rodriguez, On the Load Balancing of Edge Computing Resources for On-Line Video Delivery, IEEE Access 6 (2018) 73916–73927. doi:10.1109/ACCESS.2018.2883319.