# Experimenting with Microservice Scheduling and Performance in the Cloud

T. Dawson Lee
*Department of Computer Science*
*Vanderbilt School of Engineering*
Nashville, TN
timothy.d.lee@vanderbilt.edu

Jeerthi Kannan
*Department of Computer Science*
*Vanderbilt School of Engineering*
Nashville, TN
jeerthi.m.kannan@vanderbilt.edu

Jingyuan (Rooney) Gao
*Department of Computer Science*
*Vanderbilt School of Engineering*
Nashville, TN
jingyuan.gao@vanderbilt.edu

***Abstract*—This work extends upon research topics put forth in Fazio et al. [1]. Their paper aims to discuss open issues in microservice scheduling and performance in a containerized cloud ecosystem. To experiment with microservice-level performance benchmarking, we implemented our own Kubernetes (K8s) cluster with a Docker containerized CouchDB database microservice and ZeroMQ messaging microservice on virtual machines hosted in the Chameleon Cloud platform. Next, we ran experiments varying the number of concurrent client connections in order to capture latency. The results of this work actualize some of the issues brought up in Fazio et al. [1] and exemplify the difficulties of microservice performance monitoring in the cloud.**

***Index Terms*—Microservice Architecture, Cloud Computing, NoSQL Database, ZeroMQ, Benchmarking, Docker, Kubernetes, Ansible**

## I. Introduction

### A. Background

Microservices have the advantage of being deployed repeatedly without affecting a machine's ecosystem, and providing scalability and portability. [1] However, this comes at the expense of increased performance overhead and resource-heavy remote calls. Fazio et al. discusses open issues in microservice scheduling and performance. Specifically, the authors explore issues related to microservice monitoring, scheduling and configuration policies, and performance characterization and isolation of applications running microservice architectures.

In recent years, the microservice architecture has become increasingly popular. Dr. Sill [2] discusses the trend-setting of microservices and the future directions and paradigms which this approach will take in the coming years. A single microservice can have an isolated and unique function separate from the other microservices. Past studies have examined the performance of NoSQL databases and benchmarked performance of competing database providers [3]. By combining the open issues discussed in Fazio et al. [1] with the idea of benchmarking NoSQL databases in Kashyap et al. [3], we decided to benchmark a CouchDB microservice containerized in a Kubernetes cluster on Chameleon Cloud. Fig. 1 shows our containerized CouchDB microservice deployed in a Kubernetes pod running on a virtual machine VM2.

However, performance measurements are not always easy to gather in containerized architectures. Databases present a unique challenge given their extensive need for synchronization and orchestration when benchmarking performance [4]. Kousiouris and Kyriazis explore this issue and propose a benchmarking service tool to measure containerized and distributed databases [4]. In this paper, we measure CouchDB's containerized performance with direct concurrent client connections as well as client connections thru a ZeroMQ client-server messaging architecture (see Fig. 2). Kang et al. [5] mention common publish-subscribe software solutions and evaluate their performance under different IoT traffic conditions. In our case, we did not use the publish-subscribe pattern. We used the request-reply pattern of the ZeroMQ messaging library for communication between our clients and the containerized ZeroMQ server in the cloud. This enabled us to programmatically vary the request loads and concurrent client connections.

From the topics put forth in the aforementioned research, we implement two experiments measuring insertion and read latencies against a containerized CouchDB microservice. This work expands on Fazio et al. [1] as well as Kousiouris and Kyriazis [4]. The data gathered and implementation methods applied in our experiments show the difficulties of obtaining quality performance metrics from NoSQL databases being run within a containerized microservice architecture. The experiments' results reveal specific attributes of CouchDB and can be used to further characterize performance of containerized NoSQL databases running on virtual machines in the cloud.

### B. Rationale

We decided to tackle this project for a number of reasons. Each member of our team has previously utilized the container orchestrator Kubernetes (K8s), the container technology Docker, and the cloud provider Chameleon Cloud. We also had backgrounds deploying pipelines on the cloud with Ansible playbooks which use Infrastructure as Code (IaC). Additionally, the paper we initially chose to use as the backbone for the project was quite broad [1]. The topics in this paper and other related areas of work were consistent with the course material from our graduate level cloud computing

class. We wanted a more hands-on research project which involved deploying and coding aspects of the project. Finally, the container technologies and the microservice architecture we used in this project are widely used in the industry.

## II. Architecture and Implementations

### A. Chameleon Cloud, Vagrant, Ansible

We decided to use the Chameleon Cloud platform, a government-funded cloud provider based in University of Chicago and Texas Advanced Computing Center at the University of Texas at Austin, to host our Virtual Machines (VMs), as we had prior experience using it for previous programming assignments. We thought it would be more efficient to re-purpose the VMs we already had running on the platform rather than expend more time and effort to set up and manage VMs on other platforms such as AWS and GCP. As part of the CS 4287/5287 Chameleon Cloud classroom, we share a pool of computing resources from which we can request to create VMs when necessary and release when no longer needed. Within our classroom, we used Vagrant and Ansible playbooks to create 2 VMs, VM1 and VM2, each with the *m1.large* flavor. This flavor gives each VM 4 virtual CPUs (vCPUs), 8GB of RAM, and 40GB of memory.

We used Vagrant, VirtualBox, and Ansible to create our initial VMs, as automated creation was more streamlined and efficient than manual creation. We used Ansible playbooks in conjunction with Vagrant to have access to multiple cloud VMs at once, and perform installations and configurations through automation. Usually a significant amount of errors occur when installing software on VMs manually, so we wanted to avoid those as much as possible through automated installation. There are other technologies that we could have used to automate installations, but we preferred to continue using technologies we were familiar with. Each VM has a unique associated floating public IP to allow access from the outside world, and security groups enabled to allow traffic on ports necessary for each application, detailed in Table 1.

### B. Docker, Kubernetes

Throughout our class, we used Docker and Kubernetes (K8s) as containerized technologies, and decided to continue using them to run our microservices in containers. There are other containerized technologies out there, but we preferred to stick with the ones we had prior experience with. We use Docker to create customized container images and use open-source images from DockerHub. K8s was used to run these Docker containers within K8s pods on multiple VMs in a distributed cluster. On each VM, we used Ansible playbooks to install Docker and K8s. Afterwards, we created a snapshot of VM1 and used it to create two additional VMs with the same specifications, named VM3 and VM4. We decided to use 4 VMs for the overall architecture to measure performance and scheduling across multiple worker nodes rather than just one. All VMs had the same security groups and corresponding UFW rules enabled. VM1 was designated as the master node in our K8s cluster, with VMs 2-4 as the worker nodes, shown in Fig. 1. Each VM was able to communicate with each other through being connected in a Flannel overlay network. Scheduling was left at the discretion of the master, to run pods on any worker node it chooses.

| Security Group | Port Rules (ALLOW IPv4 from 0.0.0.0/0) |
|---|---|
| SSH | 22/tcp |
| HTTP / HTTPs | 443/tcp<br>80/tcp |
| Kubernetes (K8s) | 2379-2380/tcp<br>4040/tcp<br>5000/tcp<br>6443/tcp<br>7076-7079/tcp<br>8001/tcp<br>8080-8081/tcp<br>8285/udp<br>8472/udp<br>10248/tcp<br>10250/tcp<br>10255-10257/tcp<br>10259/tcp<br>30000-30010/tcp |
| ZeroMQ | 5555-5559/tcp |
| CouchDB | 5984/tcp |

TABLE I
Security Groups and Port Rules Used on Cloud VMs

### C. Microservices

Within our Docker containers in K8s pods, we used CouchDB as our database with which we measure insertion and read times of data, and the Python implementation of ZeroMQ to send and read data from our CouchDB database. We used CouchDB due to its open-source platform and our familiarity with it from previous assignments. For CouchDB, we used the Bitnami image found on DockerHub and ran it using Deployment and Service specification files. The pod listened on port 5984, with port 30001 exposed to the outside world. To run ZeroMQ, we created a custom Docker image and stored it on our private repository on VM1. We then used Deployment and Service specification files to run ZeroMQ, which listened on port 5556 with port 30002 exposed to the outside world. The deployment would run a Python script to initialize the ZeroMQ server, which would listen for requests from ZeroMQ clients. Clients can be run from any machine within or outside the cluster.

## III. Methods

### A. CouchDB Insertion Experiment

To measure latency of insertion times in CouchDB, we ran multiple experiments with 1, 5, 10, and 20 clients running a Python script to insert arbitrary data 100 times into a CouchDB database, measure the duration it takes to perform this operation, calculate the average of response times across 10 iterations, and save the response times and averages in a designated results file. Within a single client experiment, the script was executed from only one client and no others.

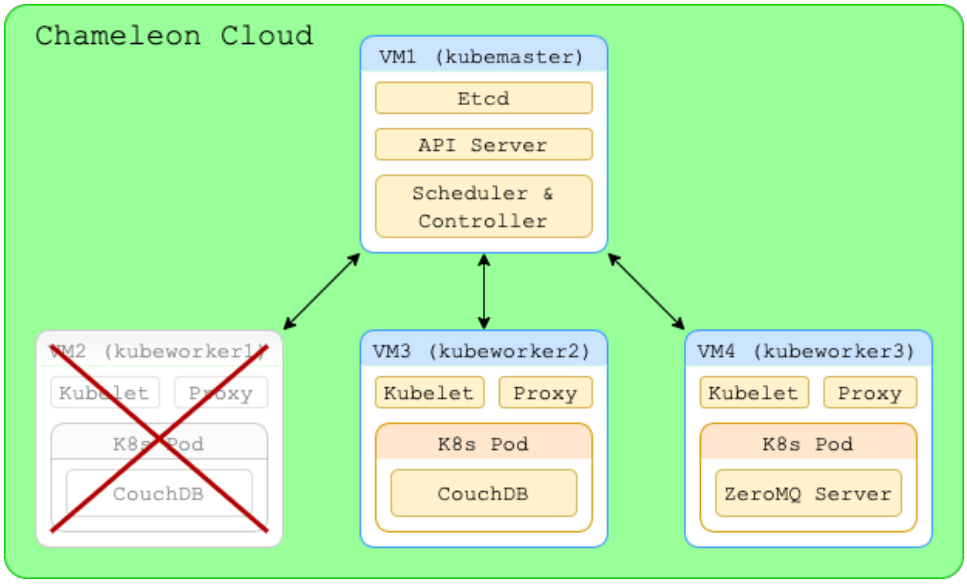


Fig. 1. Chameleon Cloud and K8s Cluster Architecture

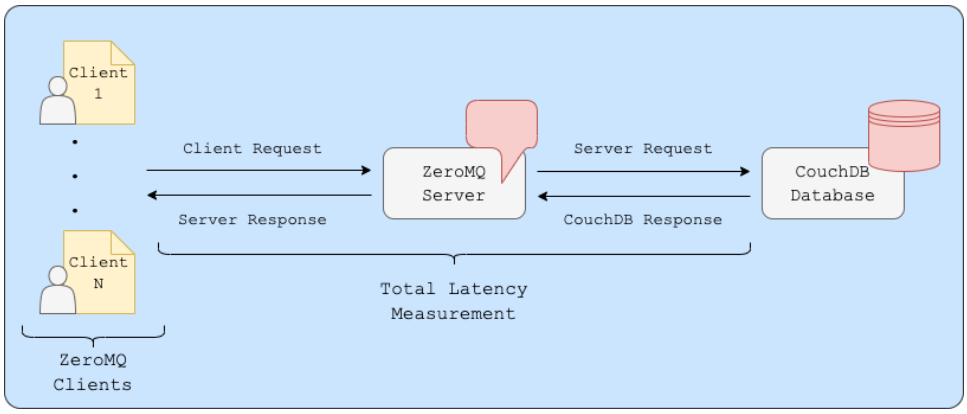


Fig. 2. ZeroMQ Client-Server and CouchDB Flow Diagram

For the 5-client experiment, we executed the insertion script from 5 different clients, and the results capture the latency that CouchDB experienced through receiving requests from 5 clients at once. The same was performed for the 10-client and 20-client experiments, but with 10 and 20 different clients, respectively, instead of 5. The results are visualized in Fig. 3.

### *B. CouchDB and ZeroMQ Read Experiment*

Multiple experiments were run to test latency of reading data from CouchDB using ZeroMQ. Clients use a Python script to send requests to a ZeroMQ server deployed on K8S. Three components are involved: ZeroMQ client app running from a local machine, a containerized CouchDB instance deployed on K8S, and a containerized ZeroMQ server deployed on K8S. For each of the 10 trials within the first experiment, the client sends 100 requests to the server. The client measures the round-trip time of each request, i.e. the time it takes from sending the request to receiving the reply from the server, including the time the server reads from CouchDB. After all the response times are stored, the client calculates the mean response time of each of the 10 iterations, calculates the total, mean, and standard deviation of response times across all requests, and saves all the results in a designated file. For the second experiment, we followed the same procedure but with 300 requests for each of the 10 trial iterations. For our experiments, we ran the ZeroMQ client Python script on 1, 5, 10, and 20 clients locally. For our single-client experiment, we ran the Python script once and waited for results to save. On 5 clients, the Python script was run almost simultaneously, and the results capture the latency that the ZeroMQ server was experiencing through receiving requests from 5 clients at once. The same was performed with 10 and 20 clients. Visualization of our average results over 10 iterations for the different experiments is shown in Fig. 4 and Fig. 5.

## IV. Issues

As we were running our experiments within the K8s cluster on our cloud architecture, our ongoing CouchDB and ZeroMQ

deployment pods started failing to communicate with each other. As we were diagnosing the issue, we discovered that one of our VMs, VM2, had disappeared. Since our CouchDB deployment was running on this particular worker node, this caused our ZeroMQ server to be unable to communicate with the CouchDB database.

Due to the nature of Chameleon Cloud resources shared among all members of the CS 4287/5287: Cloud Computing Class at Vanderbilt, all classroom members can deploy and remove resources at will, without other permissions aside from being a member of the class. As other members were deploying and tearing down their VMs in the process of developing their own final projects, our VM may have been taken down as well. The loss of VM2 and its corresponding floating IP caused a huge setback for us, as we spent much effort and time debugging the issue. Due to the limited amount of resources within our class on Chameleon Cloud, it was not possible for us to bring VM2 back up, and the floating IP was assigned to another group's VM. This highlights the limitations of using a Cloud platform that is shared among other members, with a lack of restrictions and security policies to protect resources from unintentional deletions.

After VM2 was deleted, our K8s cluster marked the node as *NotReady*, and the deployment rescheduled the pod to VM3. Hence the IP address of the CouchDB server changed to the floating IP assigned to VM3. Our existing data within our previous CouchDB database got deleted, making the CouchDB database in VM3 a clean slate. The CouchDB deployment pod originally on VM2 went to a perpetually *Terminating* state. Hence, we had to forcefully delete the previous CouchDB pod before we could cordon, drain, and delete VM2 from the K8s cluster nodes. When we applied the CouchDB deployment again, we noticed that it's not only accessible using the floating IP of VM3, but the floating IP of VM1 as well. We are not sure what causes this behavior. Perhaps there is port forwarding from the master node to the worker node. Our ZeroMQ server and client were still unable to retrieve data from our database after CouchDB was deployed again. These issues caused us to lose time towards running additional experiments with ZeroMQ and Stress-NG. To fix these issues, we reset our K8s cluster, purged the K8s installation and dependencies from all our VMs, reinstalled K8s on our VMs, and re-ran the cluster. After applying our service and deployment CouchDB and ZeroMQ pods, communication between ZeroMQ clients, the ZeroMQ server, and CouchDB was restored.

We originally planned on using Stress-NG to stress-test our pods and compare performance of other microservices under stress with those of normal conditions. However, due to the time and effort expended to salvage our cluster and experiments, we were unable to do so. We hope to utilize Stress-NG for further experiments in future work.

From this setback, we were reminded of the issues that come with using a shared pool of resources from a cloud provider, and for the future, we hope that Chameleon Cloud implements security protocols to prevent this issue from happening again. Perhaps the cloud provider can implement a passphrase requirement whenever a request to delete an instance is initiated, with only the classroom admins having the ability to override this requirement. We think this would prevent unintentional automated and manual deletions of VMs from a Chameleon Cloud classroom, and give students more security over their requested resources.

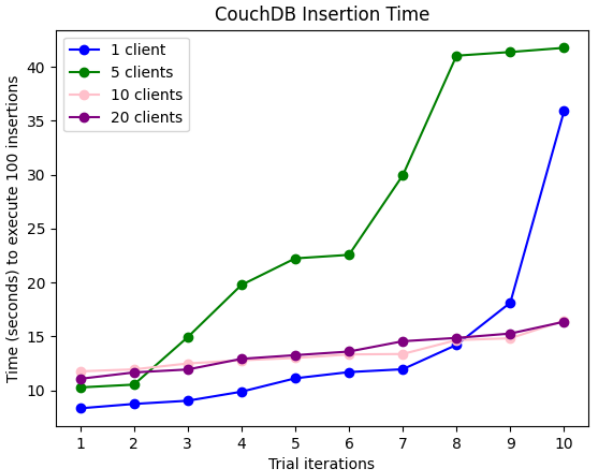


Fig. 3. CouchDB Insertion Time Latency


Fig. 4. CouchDB Latency via ZMQ (100 reads / trial)

## V. Results and Discussion

To measure insertion times of 100 points of data in a CouchDB database, we ran multiple experiments with 1, 5, 10, and 20 clients. We expected that the latency of insertion times would increase as the number of clients increased. However, as shown in Fig. 3, response times with 10 and 20 clients were similar to those with 1 client. Whereas the response times with 5 clients were significantly higher than

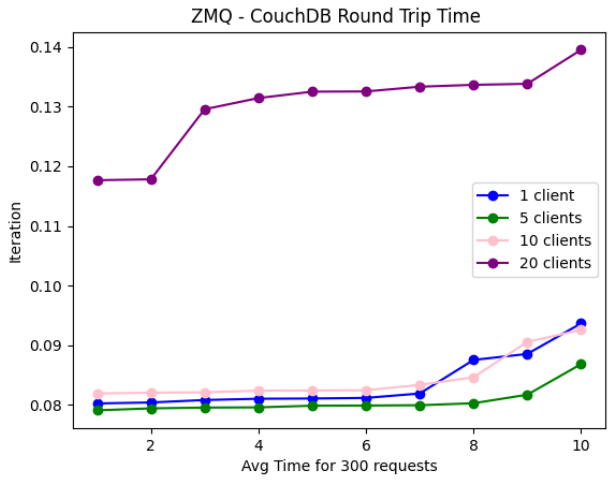


Fig. 5. CouchDB Latency via ZMQ (300 reads / trial)

the others. At the 10th iteration, the single client experiment and the 5-client experiment had significantly higher response times: about 35.9 and 41.8 seconds, respectively. Whereas the 10-client and 20-client experiments had 16.5 and 16.4 second response times. We can see that the performance scales relatively well across the 1-, 10-, and 20-client scenarios. Since the client connects directly to the CouchDB database, we believe that these results are due to the Database Management System (DBMS) optimization mechanism CouchDB uses to schedule concurrent write transactions. For a commercial-capable database, 20 concurrent connections are likely not impacting the performance in terms of latency. Based on this conjecture, we suspect that the 5-client performance results are likely due to networking issues or hardware resource contention occurring on the node on which CouchDB is deployed.

For the CouchDB read via ZeroMQ experiment, we first notice that while the round trip time tends to be longer as the number of concurrent clients increases, it scales very well with respect to the number of concurrent clients, at least for 1, 5, and 10 clients. Note that ZeroMQ handles multiple clients by fair-queuing clients whenever it receives a request and replying to the client polled from the queue whenever the send procedure is invoked. The round trip time includes the time to schedule and queue the clients, read data from CouchDB, send data back to the client, and network over both communications. With 20 clients, we believe that the significant time increase is due to the scheduling overhead from multiple concurrent connections. As for the database performance, we observe that the number of reads per trial iteration does not have a huge impact on the round trip time. We believe that this is because the client requests need to be scheduled and sent to CouchDB via the ZeroMQ server instead of directly sent to the database all at once. This means that because the ZeroMQ server sitting in front of CouchDB has performed a round of scheduling prior to the request being sent to the database, CouchDB is not experiencing significant scheduling overhead increase as the number of concurrent client requests increases. Another reason could be that, in general, read operations are cheap and allow the DBMS to perform less restrictive scheduling of concurrent transactions.

## VI. Teamwork

Throughout this final project, we managed to split the workload among all 3 project members. We worked together both virtually using Zoom, and asynchronously using Slack for communication. To manage version control, we used a shared private Github repository. VM creation was done by all 3 project members using Ansible playbooks and the image snapshot. Installation of Docker, K8s, and other necessary technologies for the VM itself was performed by Jeerthi using Ansible playbooks. Jeerthi copied files to the VMs, modified and set up the K8s cluster, and ran all the necessary K8s pods using Ansible playbooks. Dawson and Jeerthi debugged and modified the K8s cluster setup and pods when VM2 disappeared. Rooney created Dockerfiles, K8s specification files, and Python scripts for the ZeroMQ server and client experiments where the latency of reading data from CouchDB was measured and used for subsequent calculations. Rooney and Dawson created and edited the Python script which inserts data into a CouchDB database and measures the latency of insertion times. Jeerthi and Dawson worked on K8s specification files for CouchDB and ZeroMQ. Dawson and Rooney tested and ran the experiments to measure insertion and read times with CouchDB and ZeroMQ. All 3 team members worked on the final report: Dawson and Rooney made the figures, and all 3 team members wrote and edited the text.

## VII. Conclusion

In this project, we researched and analyzed some of the factors impacting microservice scheduling on the cloud, and we investigated some issues relating to the topic by building and experimenting with our own microservice setup. We implement Infrastructure as Code and extensively use it to build the cloud computing setup, including resource provision, data transfer, and containerization and deployment of microservices. According to our research, the variability of application implementation details and use cases is one significant factor of the difficulty of containerized microservice scheduling. We designed and conducted experiments to measure the latency to a containerized CouchDB service for write operations via direct client-database connection and the latency for read operations via a client-server-database connection using ZeroMQ. We find that for the write operations via direct connections to the database, the latency scales well with most numbers of concurrent connections and we suspect that this is due to database-side scheduling optimization. For the read operations via a ZeroMQ server, we find that the performance scales well with respect to the number of read queries per request. However, it does tend to degrade as the number of concurrent connections increases. We think that this is because

most of the scheduling is done at the ZeroMQ server sitting between the client and the database, and its overhead impacts the performance. We also observe outliers throughout our experiments, and we believe that they occur due to networking issues often found in cloud-based systems.

Our project mostly focuses on the performance metrics of various implementations and use cases of cloud-based applications involving database operations. However, we did not investigate into virtual resource management, which is also an important factor contributing to the difficulty of building efficient systems of containerized microservices. The scale at which we conducted our experiments is also rather small, due to limited time and computing resources. Implementing the experiments at a larger scale would have enabled us to show more interesting behaviors and uncover more issues with containerized microservices.

Utilizing Stress-NG to observe and measure the effect of stress-tests on performance of insertions into CouchDB and reading data from CouchDB, is a direction for future work.

Our repository with all our code for the final project is located on GitHub: https://github.com/jeerthik/CS5287-FinalProject

## REFERENCES


[1] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen and M. Villari, "Open Issues in Scheduling Microservices in the Cloud," in *IEEE Cloud Computing*, vol. 3, no. 5, pp. 81-88, Sept.-Oct. 2016, doi: 10.1109/MCC.2016.112.

[2] A. Sill, "The Design and Architecture of Microservices," in *IEEE Cloud Computing*, vol. 3, no. 5, pp. 76-80, Sept.-Oct. 2016, doi: 10.1109/MCC.2016.111.

[3] Kashyap, Suman, et al. "Benchmarking and Analysis of NoSQL technologies." *Int J Emerg Technol Adv Eng* 3.9 (2013): 422-426.

[4] Kousiouris, George, and Dimosthenis Kyriazis. "Enabling Containerized, Parametric and Distributed Database Deployment and Benchmarking as a Service." Companion of the *ACM/SPEC International Conference on Performance Engineering*. 2021.

[5] Kang, Zhuangwei, et al. "Evaluating DDS, MQTT, and ZeroMQ Under Different IoT Traffic Conditions." (2020).