COLORADOSCHOOLOFMINES
EARTH • ENERGY • ENVIRONMENT

CSCI 370 Final Report

ICR Cloud Messaging System

Camden Fritz
Massimo Giusti
Viktorija Mikelevicius

CSCI 370 Fall 2022

Ms. Kelly

Table 1: Revision history

Revision	Date	Comments
New	8/25/2022	Initial notes on parts I-III
Rev – 2	9/17/2022	IV-VI, finalizing design document
Rev – 3	10/20/2022	VII-VIII, Software quality plan and Ethical Considerations
Rev – 4	11/13/2022	IX-XII, Results and the start of Lessons learned and profiles
Rev – 5	11/29/2022	Updates and additions to all sections
Rev – 6	12/6/2022	Final revisions

Table of Contents

I. Introduction

We are the team working for ICR testing different microservice messaging systems. ICR is an employee-owned defense contracting company that was founded in 2014 by two guys that had to leave a company in the same field, so they decided to create their own. They work with the Department of Defense and other members of the Defense and Intelligence communities, focusing on solving difficult problems quickly and reliably. ICR has buildings and employees all over the country and the world, but their first and largest division is in Colorado.

Since ICR deals with classified information, we were not told what system or what kind of data they were going to be sending between the microservices, but we did not need to know that in order to test the messaging systems. These systems are common and an industry standard. Most of the experiences we have with code in school are working with one script that does all we need it to do. When working with a real-life system that will be running long-term, it will be broken down into different microservices. This prevents the creation of a monolith piece of code that is impossible to change and test. But when we have microservices, they need to send and receive data from each other. When there is a large system and lots of data, keeping track of where everything needs to go and making it reliable becomes more complicated. Messaging systems introduce a broker that sends messages to a queue or topic that a microservice can subscribe to and receive the data it needs. It adds another layer of abstraction, since a service doesn't need to know where it is sending data or where it is receiving data from.

This project entails exploring a few different cloud agnostic microservice messaging options to provide information that we will use to help craft a more flexible solution. The goal was to find the best messaging solution from a list of commercial products available. This project includes research at the beginning into the

available products, implementation of selected technologies, and then testing between different products based on resilience and load capability.

II. Functional Requirements

Our functional requirements include:

- Creating a proof of concept for each messaging system that seamlessly transfers information between different microservices
- The final documentation having thorough performance testing
- The messaging system needs to:
    - send all messages to all the receivers
    - handle multiple containers
    - preferably be able to still send messages after a receiver shuts down with no data loss

Since the sample code has been given to us, we did not need to develop data producers, senders, and consumers that can work together to transfer data.

III. Non-Functional Requirements

We will need to take into consideration that the services we test and recommend:

- Are easy to use
- Have good documentation
- Have superior performance
    - The service needs to pass the integration tests, and the one we recommend to ICR needs to perform better than the rest based on the priorities they give us
- Can deploy within Docker
- Can use Java and Python
- Has security capabilities

IV. Risks

Some anticipated challenges were:

- Not making a decision by the end of the semester
    - This could be as a result of trying too many technologies and not finishing our testing.

- Not having enough work to do or demo by the end of the semester
    - Since this project is just configuring and testing certain services, it was fair to worry that if we had a lot of time to work on the project, we wouldn't have anything to do at the end of the semester.
- Picking technology in the beginning that is too hard to install or test
- Not being able to work for extended periods of time, either for other school responsibilities or illness

V. Definition of Done

The final deliverable for this project is having at least 2 working prototypes of messaging technologies using the code provided by Mike and Daniel from ICR as a jump starter. In addition to the code in our repository, we provided a recommendation on which product we think is best supported by stats, documentation, and experience when using the product.

VI. System Architecture

Technical Design Issues:

- Connecting the given generators and receivers effectively with the implemented messaging system.

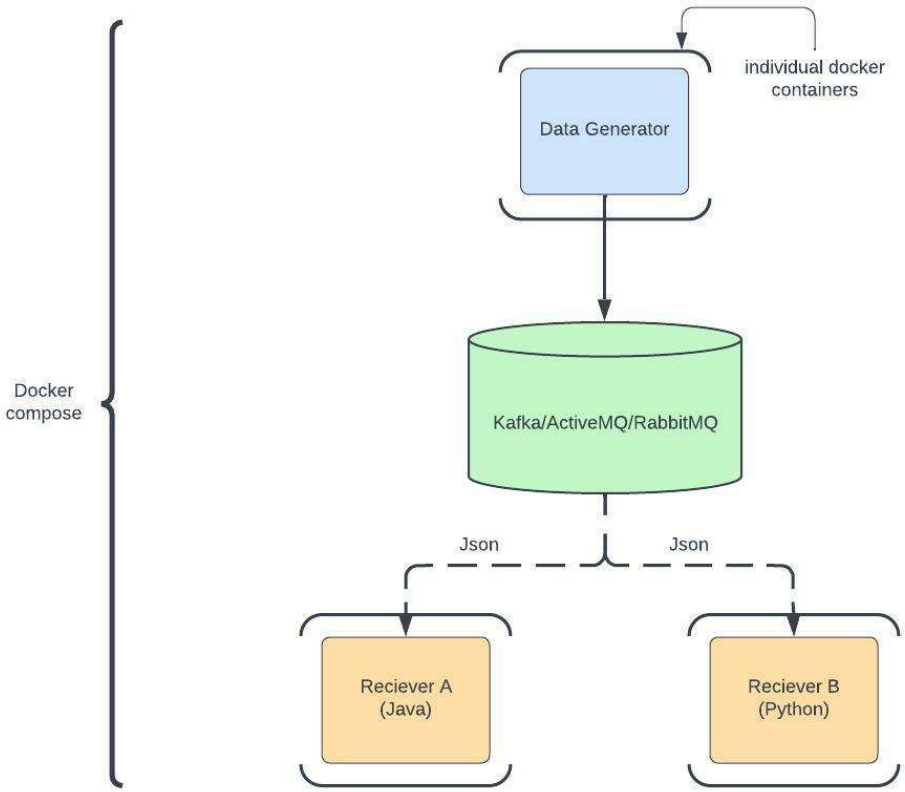

Figure 1: Given system architecture

This initial design given to us by the client includes one data generator and two receivers all in individual docker containers. Our job was to incorporate the messaging system as the in-between step that takes data from the generator and sends Json to the receivers. Everything is included in one docker compose image.

The data generator is composed of a producer and a sender. In the sample code provided to us, the generator creates data in the form of “Vehicles” that are then sent to the receivers. Each of the messaging systems redirects the data sent to a topic or queue. The receivers are connected to or subscribed to these topics or queues and the data is transmitted to each of the receivers even though they are written in different languages and use different methods to receive that data.

Considering what we have, the goal of our integration and performance testing was to take this structure, which is the most basic structure that still adheres to our requirements, and make it more complex with the same components. The tests are done by creating new docker compose files that spin up more data

generators, more receivers, more queues/topics, and/or more messages.

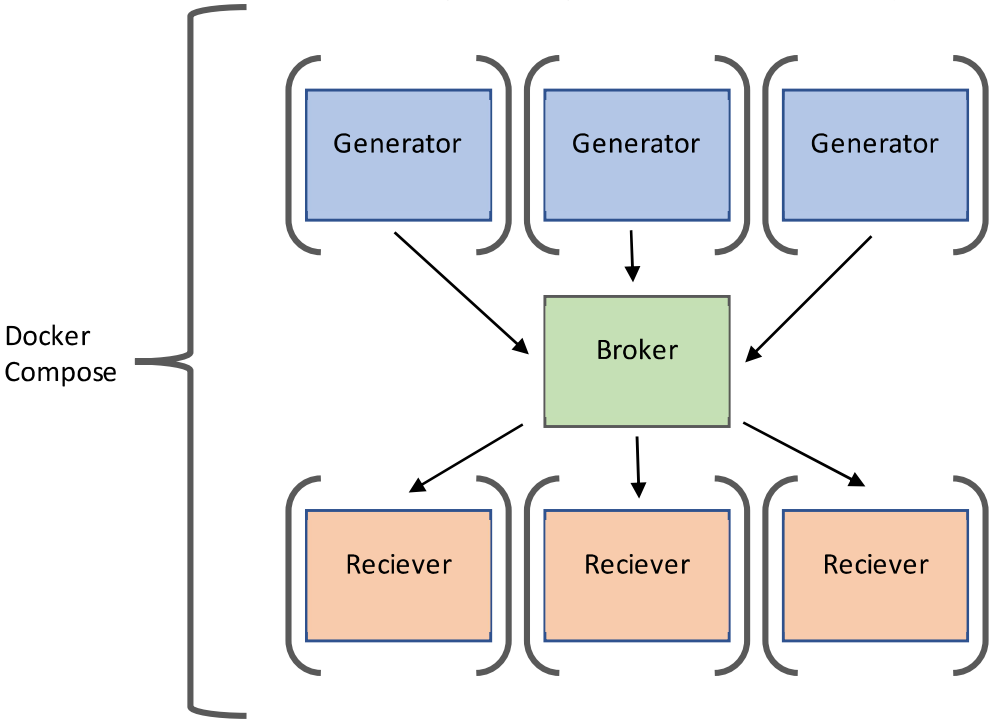

Figure 2: System Variation #1

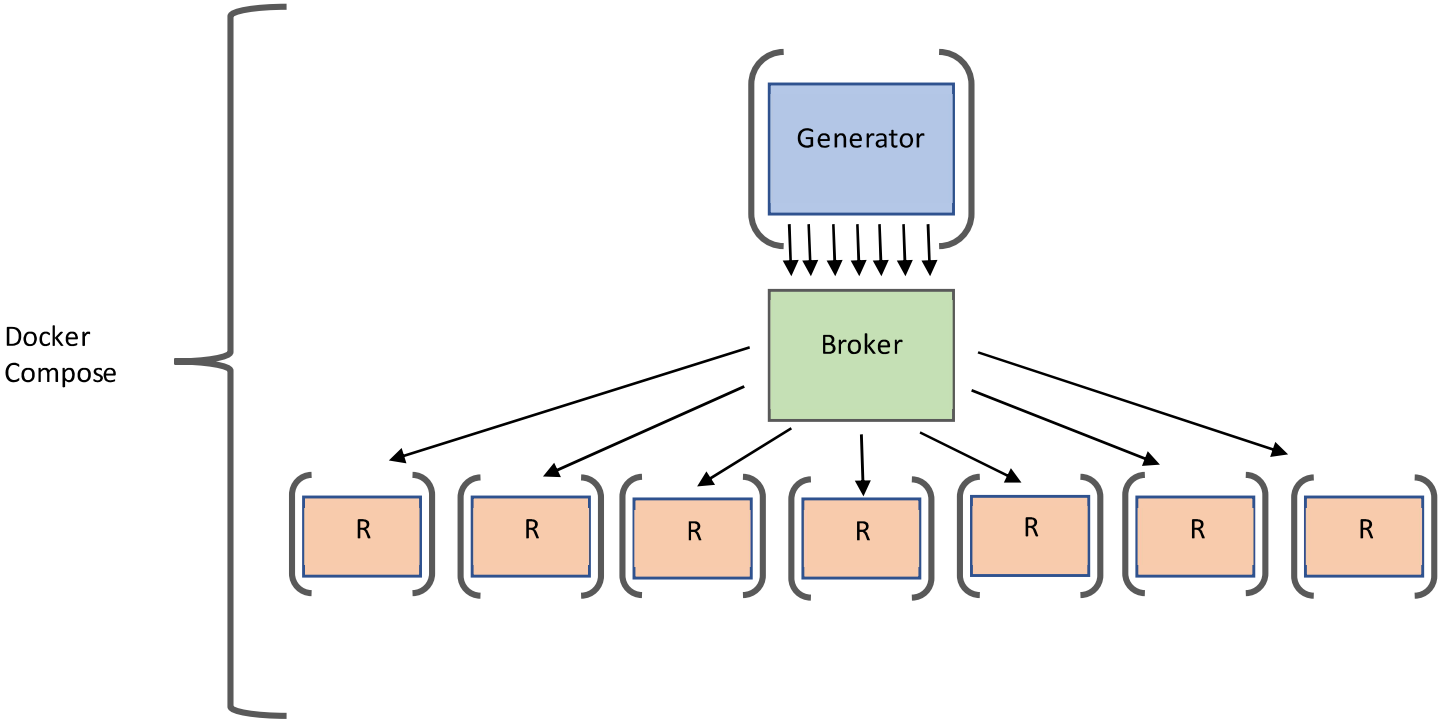

Figure 3: System Variation #2

And there could be countless other variations using the same microservices.

Technical Design:

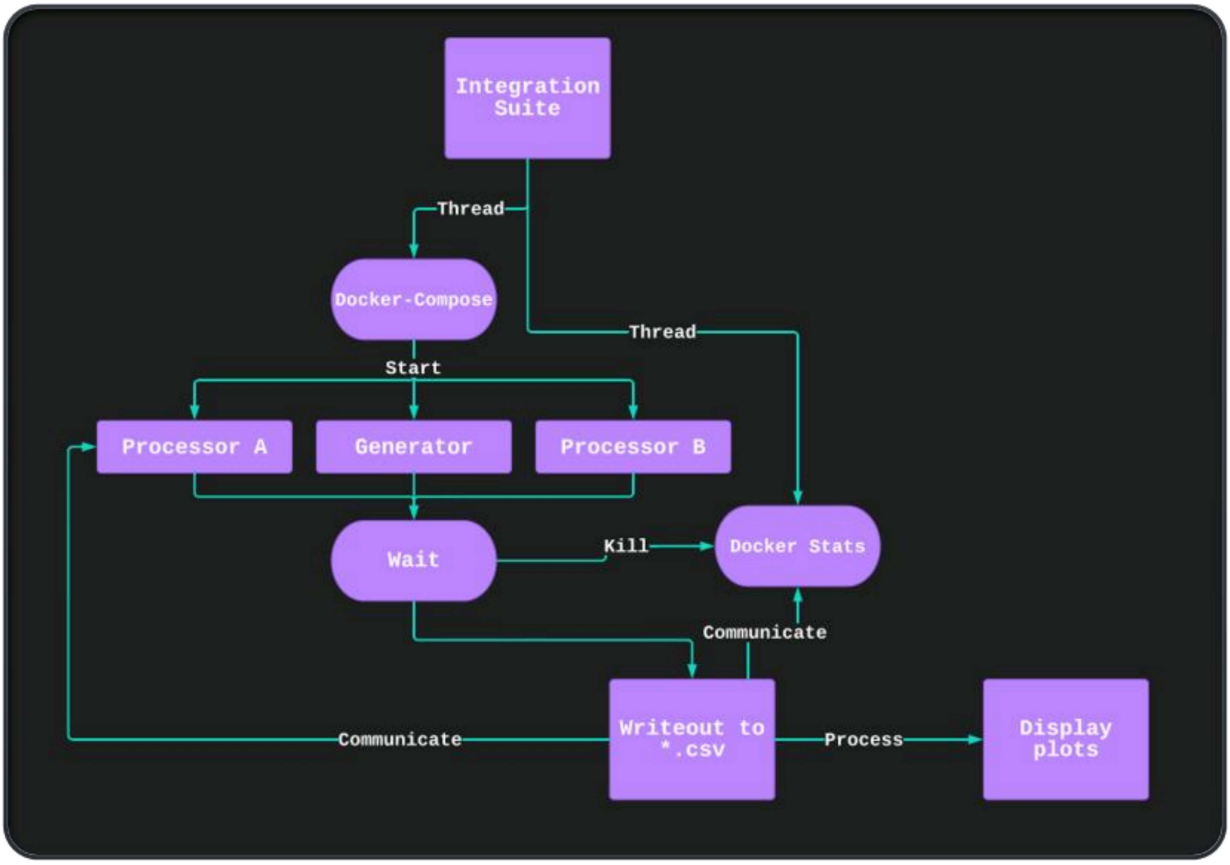

Figure 4a: Integration Suite Process

The integration test suite is used to deploy the instances of microservices onto a container while simultaneously aggregating performance data on each container. This is done on two separate threads that communicate and run the tests under the configurations specified and then display a visualization of the resulting performance data. We can configure a test by using command line arguments in the python script which modify the messages sent at a time, the broker that we are running the test for, the path of the docker-compose file of the microservices, the time we are running the test for, and finally, the interval at which we send messages.

A test can be configured from the command line using command line arguments. Not all arguments need to be set and this logic is controlled using Python's argparse package. The arguments allow you to define two distinct types of tests. A duration test runs for a specified amount of time sending a consistent load of messages on intervals. Arguments that must be set for this are num-messages which define how many messages to be sent at a time, time which declares the duration of the test in seconds and message-rate which is the interval at which the send command is called. The other test we can define is a scaling test that declares

how many messages a producer needs to make and then how many messages the consumer expects to receive. This test is defined using the num-messages to declare the total number of messages to be sent and not setting time since we do not want to limit the duration of the test.

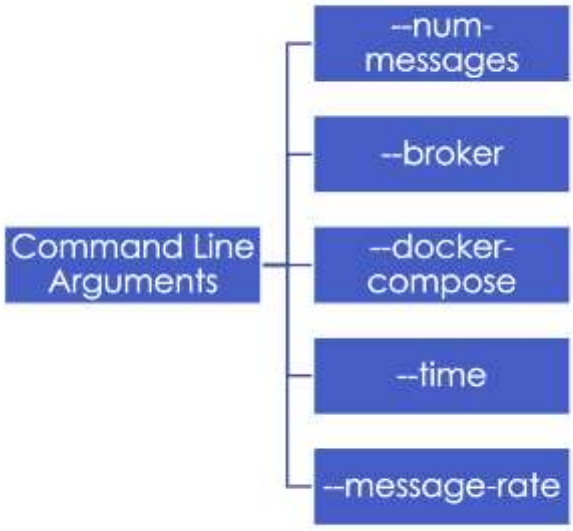

Figure 4b: Command Line Arguments

VII. Software Test and Quality

Our project can accurately be described as a research project testing different messaging systems, so testing the code and what we made is the end deliverable. The testing we did on our messaging systems can be divided into two categories: Reliability and Performance Testing.

Reliability Testing:

This is the category where we pushed the limits of the messaging system and see how much it can handle before the usability is impaired significantly. This includes tests such as:

- One producer to one consumer
- Many producers to one consumer
- One producer to many consumers
- Increasing message size
- Consumers shutting down and still receiving messages when they start again

However, we were only able to thoroughly test reliability by shutting down a broker or service and making sure the messages are still received. We focused on this test because the client specifically requested it and communicated that it was a high priority.

Performance Testing:

This category is where we were able to come up with numbers to accurately compare the different messaging systems. This included tests such as:

- How many of the above reliability tests they passed
- How many messages can be sent
- How long it takes for a certain number of messages to be sent
- How long it takes for a certain number of messages to be received
- How many containers we can get to communicate with each other

VIII. Project Ethical Considerations

These are the specific ethical points we have considered over the course of this project. (All labeled points are from the IEEE Code of Ethics)

Quality of our Work:

- 3.01. Strive for high quality, acceptable cost and a reasonable schedule, ensuring significant tradeoffs are clear to and accepted by the employer and the client, and are available for consideration by the user and the public.
- 3.02. Ensure proper and achievable goals and objectives for any project on which they work or propose.
- 3.05. Ensure an appropriate method is used for any project on which they work or propose to work.
- 3.07. Strive to fully understand the specifications for software on which they work.
- 3.08. Ensure that specifications for software on which they work have been well documented, satisfy the users' requirements and have the appropriate approvals.
- 3.09. Ensure realistic quantitative estimates of cost, scheduling, personnel, quality and outcomes on any project on which they work or propose to work and provide an uncertainty assessment of these estimates.
- 3.10. Ensure adequate testing, debugging, and review of software and related documents on which they work.
- 3.11. Ensure adequate documentation, including significant problems discovered and solutions adopted, for any project on which they work.

One of our main priorities is to produce quality work for our clients that they can use after we are finished working on it. This includes practices such as communicating realistic expectations and accurate progress reports, thoroughly documenting all our progress, and performing and documenting our software tests (as described in section VII: Software Tests and Quality).

Employee and Client Relations:

- 2.01. Provide service in their areas of competence, being honest and forthright about any limitations of their experience and education.
- 2.02. Not knowingly use software that is obtained or retained either illegally or unethically.
- 2.05. Keep private any confidential information gained in their professional work, where such confidentiality is consistent with the public interest and consistent with the law.
- 2.06. Identify, document, collect evidence and report to the client or the employer promptly if, in their opinion, a project is likely to fail, to prove too expensive, to violate intellectual property law, or otherwise to be problematic.

Since we are working with a small team of professional software engineers, we want to be professional in our communication with them and what we say about them and their company. We need to be transparent about any software we use or any problems we come across regarding our knowledge or the project itself.

Teamwork:

- 7.03. Credit fully the work of others and refrain from taking undue credit.
- 7.04. Review the work of others in an objective, candid, and properly-documented way.
- 7.05. Give a fair hearing to the opinions, concerns, or complaints of a colleague.

Since this is a team project, we need to treat our teammates with respect by giving credit to where it is due and professionally giving and receiving constructive feedback.

IX. Results

Testing is a large part of this project so we had to take a lot of time to plan out how we would test our messaging systems. The first tests are integration tests that change the configuration of the system by adding more messages all at once or in batches. We also tested not only if the systems can handle these configurations, but how well they do so.

Performance Tests:

(Listed in the order of ActiveMQ, RabbitMQ, Kafka, then a comparison between all three)

Sending 1000 messages:

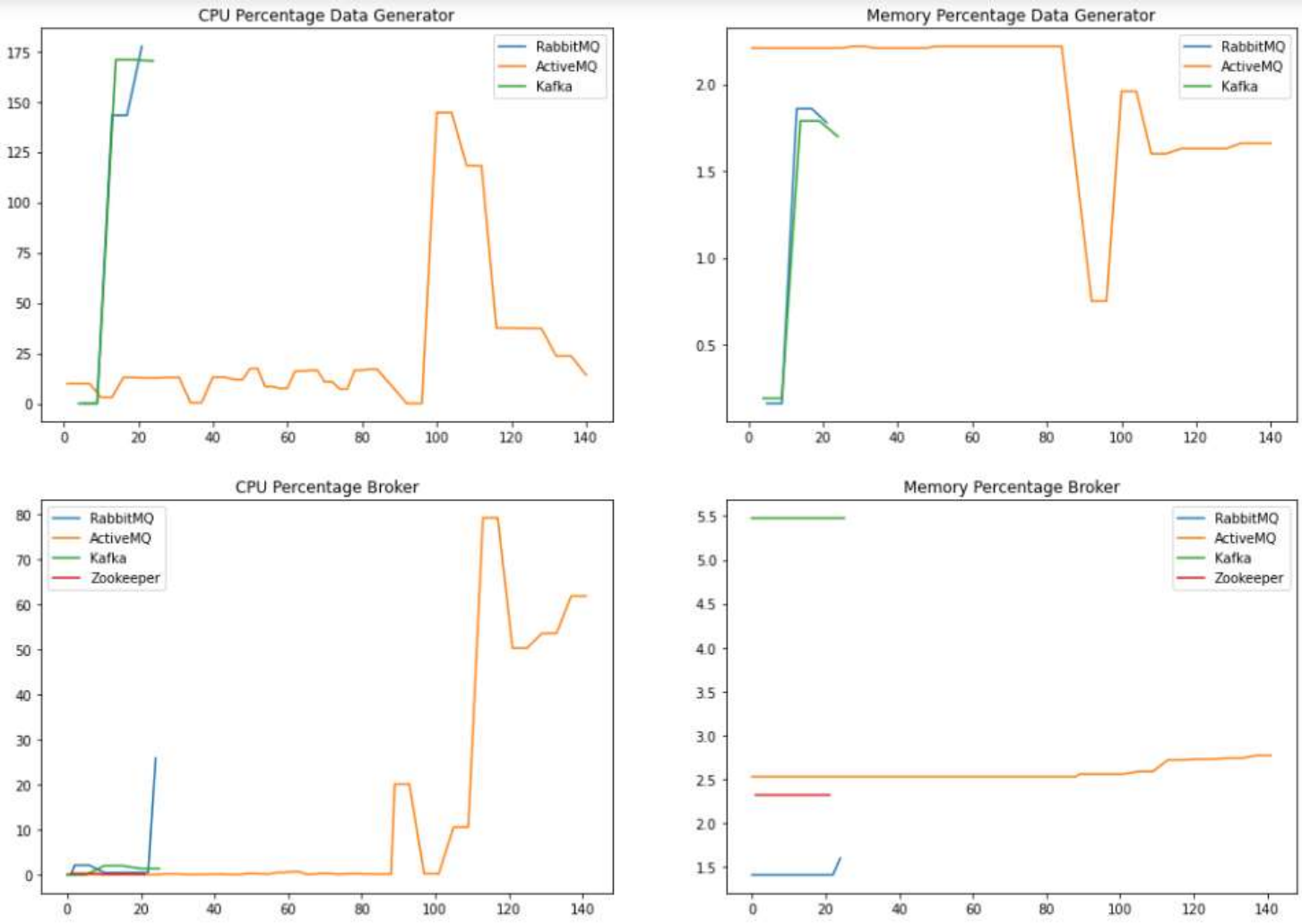

Figure 5: Scale Test – 1000 messages

Observations: ActiveMQ had the lowest peak CPU usage while Kafka had the highest. RabbitMQ had the lowest peak memory usage while Kafka had the highest. ActiveMQ had the largest gaps and changes in CPU and memory usage while Kafka was the most consistent. ActiveMQ took significantly longer than either of the other services.

Sending 1000 messages with 1000 delay and 1500 duration:

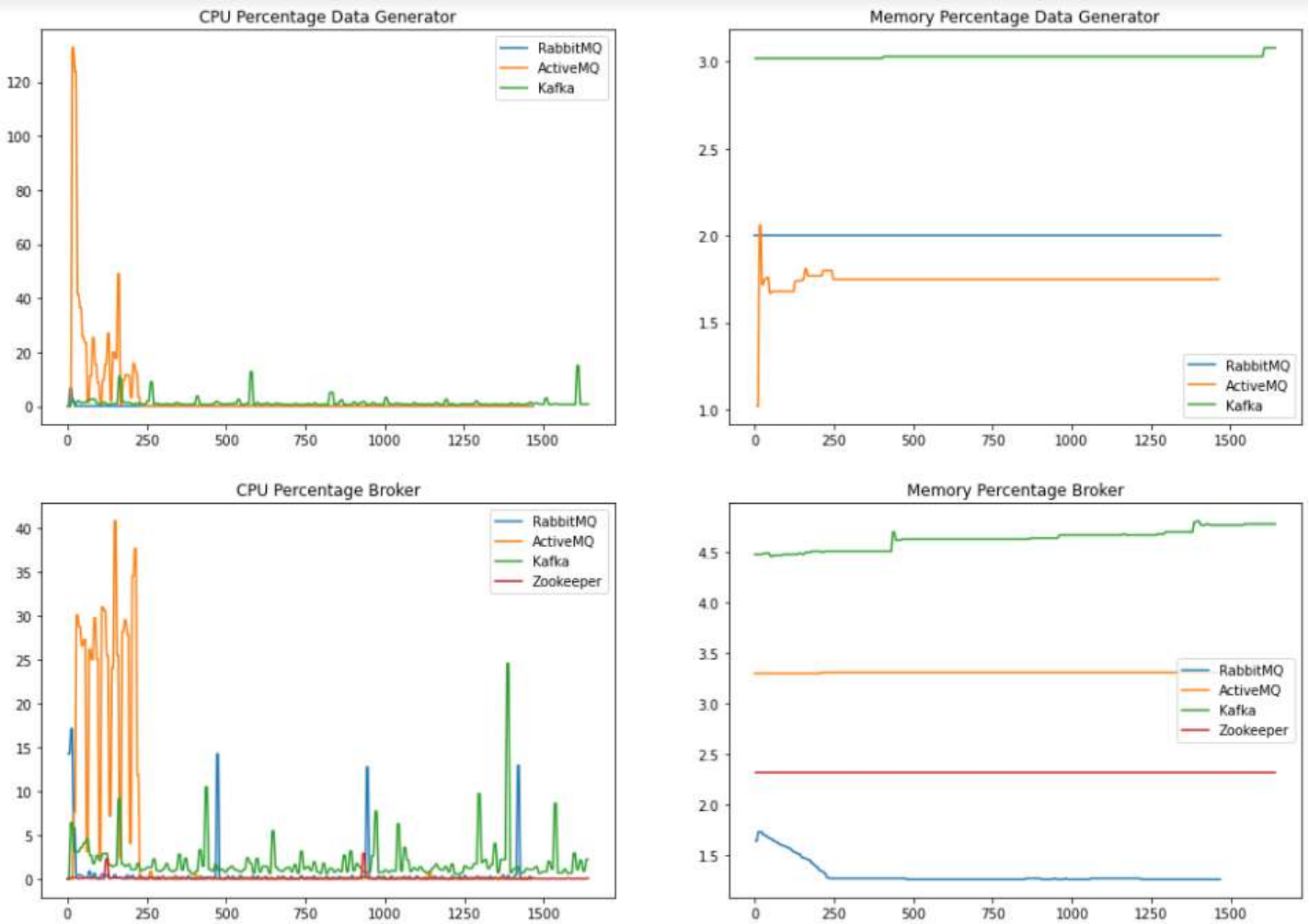

Figure 6: Duration Test

Observations: The peak CPU percentage increased for ActiveMQ but decreased for the other two. Although, ActiveMQ had a large spike at the beginning which then flattened out at a lower CPU percentage. RabbitMQ had data-processor-b leveling at about 100 and the rest leveling out at about 0 just like ActiveMQ, but RabbitMQ didn't have the same amount of variability at the beginning with the occasional spike in usage from the other data processor and the broker. Kafka has a lot more spikes of CPU usage, but the peak is significantly lower than the other two and lower than any other test. The memory graphs are all consistent with each component leveling out at a certain memory percentage. RabbitMQ has the smallest peak memory usage while Kafka has the largest.

Sending 100,000 messages:

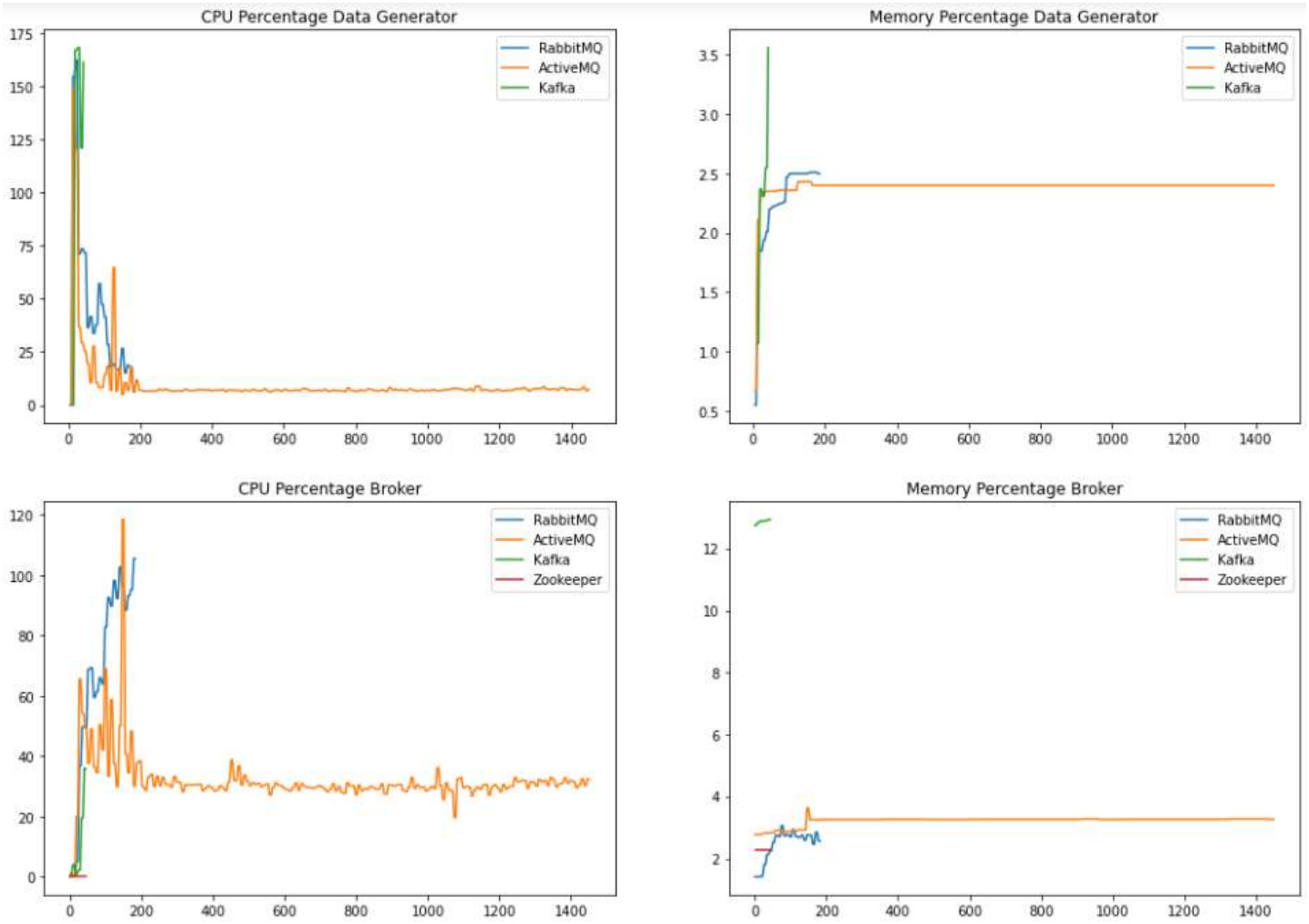

Figure 7: Scale Test – 100,000 messages

Observations: Both ActiveMQ and RabbitMQ have large spike of CPU usage at the beginning of the test, however while ActiveMQ gradually leveled out RabbitMQ did not. Kafka had the largest peak CPU and memory usage, but the Kafka broker seems to be an outlier when using the most memory while its other components have memory usage comparable to the other two systems. ActiveMQ had the smallest peak CPU usage while RabbitMQ had the lowest peak memory usage. RabbitMQ and Kafka finished quickly, but ActiveMQ did not.

Sending 1,000,000 messages:

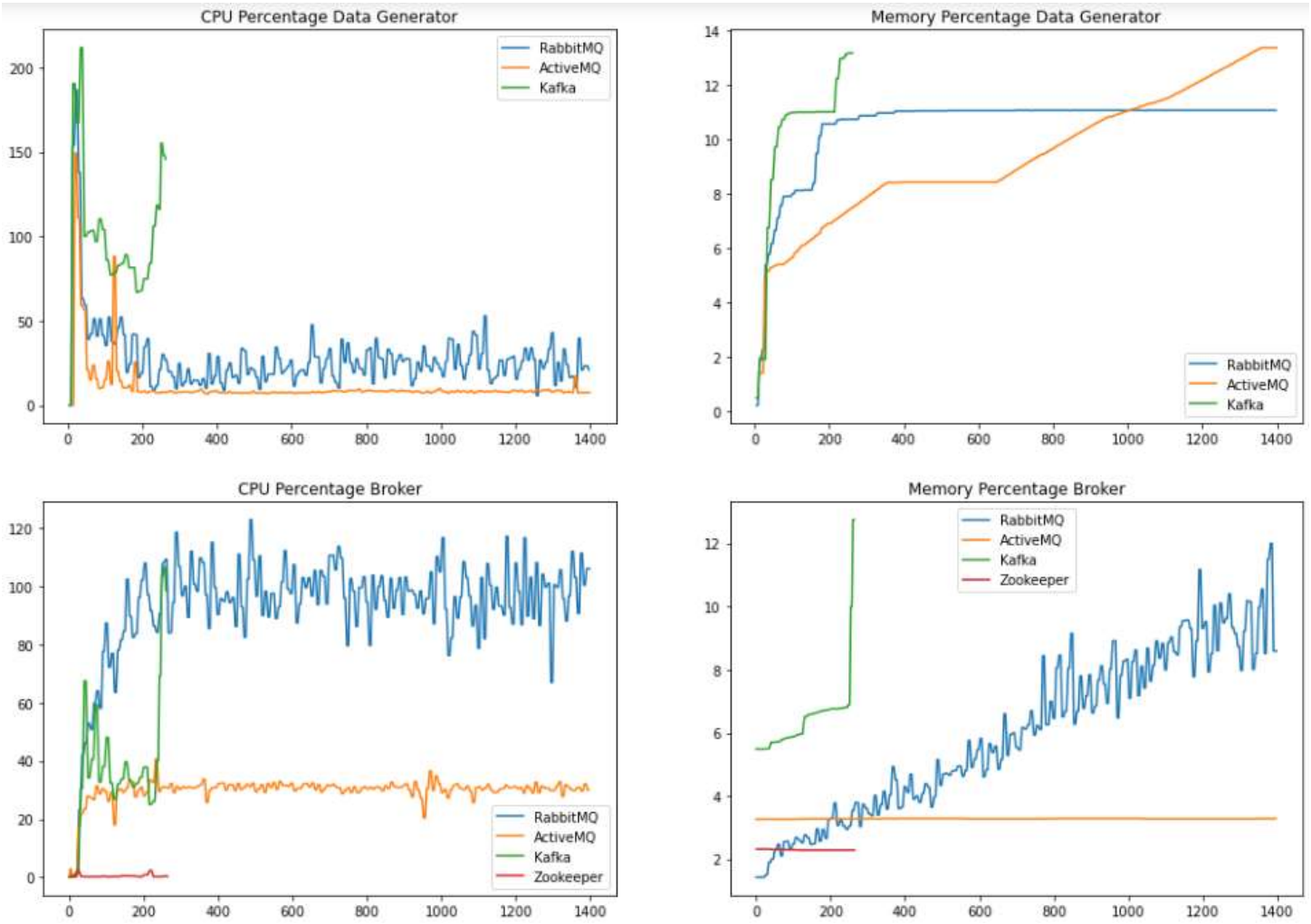

Figure 8: Scale Test – 1,000,000 messages

Observations: ActiveMQ seems to have a very similar pattern of CPU usage to when it was sending 100,000 messages with only a slightly higher peak CPU percentage. RabbitMQ also has a similar pattern but with more inconsistency and a higher peak CPU percentage. Kafka has a significant spike in the beginning of the test for the first time, is a lot more variable and inconsistent than the other tests, and the highest CPU usage of the three. ActiveMQ had consistent memory usage except for the data generator that was steadily increasing as the test went on. The RabbitMQ broker seemed to have increased its memory usage at an alarming and inconsistent rate that didn't level out at the end of the test. ActiveMQ had the largest peak memory usage with RabbitMQ having the lowest peak. However, from the last test with only 100,000 messages, both ActiveMQ and RabbitMQ increased their memory usage up to a similar peak Kafka had in both tests. Both RabbitMQ and ActiveMQ did not finish, but Kafka did only slightly longer than the 100,000 messages test.

Summary: There are exceptions, but the clearest patterns that emerged from these tests are that ActiveMQ is the most variable with the resources it uses, and Kafka generally uses the most resources. RabbitMQ does well with lighter loads, it is ideal if the number of messages and strain on the system will be smaller. Kafka is more

reliable and scalable since it is the only one that completed all the tests, and it is clear from the second test with delays that when configured to its strengths, the amount of resources Kafka uses can be managed.

Integration Tests:

ActiveMQ and RabbitMQ:

The integration tests prove all services to be reliable under extreme pressure with consistent message flow from a producer. Brokers have handled the distribution of over 200,000 messages consistently being sent to two different consumers. RabbitMQ and ActiveMQ are both JMS listeners, so their implementation is similar but how they run is different. Under the test previously mentioned RabbitMQ processed messages significantly faster than ActiveMQ, but this came at the cost of higher memory and CPU percentage on the containers. The broker also seemed to scale its memory usage as time went on where ActiveMQ stayed consistent for the entirety of the test.

This project introduced a lot of new technology which took time to pick up. The largest takeaway was the understanding of how messaging systems work, and how to implement them and configure them to the test requirements. Docker and Docker Compose are also valuable skills we learned, containerizing processes and then deploying them in a testable fashion.

Kafka:

Overall, we've done lots of performance testing with Kafka as configurations scale, starting with looking at CPU and MEM% with a 1-1 with one broker and eventually ran more tests with more micro services, more brokers or both. We also tested reliability during outages. We tried to see what would happen in a multi-broker configuration if one instance of a broker went down and the result was that messages still went through. In addition, we tested what would happen to a configuration if a consumer went out. What would happen in that case was that when the consumer was out, the queue in the broker grew and when the consumer booted back up, it received all the messages in the queue all at once without data loss.

Also, we've tested what would happen to performance as brokers scale. Since topics are replicated for each instance of the broker, the mem% stayed the same, but overall CPU% was lower. Also, one thing we noted is that startup is very intensive for the broker itself. Trying to connect micro services at the very beginning and creating topics creates a spike in CPU%.

X. Future Work

The main way this project will be continued is when ICR comes to their own decision as to which messaging system they prefer and implement it. However, further testing could be beneficial since we did not have time to cover everything.

Other tests include:

- Testing different configurations
    - Producing multiple docker compose files
    - Increased amount of containers
    - Increased number of queues and/or topics
- Manually configuring a cluster (with Kubernetes) and fully scaling the service.
- Increasing the size of the messages
    - Was not part of our requirements, but this test was recommended to us by our advisor because of her experience with Kafka having trouble with larger messages.

XI. Lessons Learned

- As a group, we first had to understand the sample code that the clients gave us. We had to play with the code, make mistakes, and get used to these new programs and technologies. We had to do research on how to use Docker and Spring not only to get the sample code working, but to modify it.
- Through Docker metrics and scaling the messaging systems, we learned how to efficiently and effectively test the performance of Kafka, ActiveMQ, and RabbitMQ.
- We learned how to interact as a team, the best example being when preparing for presentations since a lot of research was individual. When preparing presentations, we learned to bounce ideas off each other and give constructive feedback in order to make a cohesive and well-rehearsed presentation.
- We learned how to pivot when things don't go as planned. Due to system incompatibility, implementing RabbitMQ proved to be a different type of challenge compared to what we had anticipated. As a result, we had to adapt and shift around responsibilities.
- Our client is a part of a real company outside of Mines, so our communication and meetings had to be intentional. We learned how to update our client regularly, ask questions when we could, and keep in touch when communication was difficult on the client's end.
- How to write code that acts the same way with different technology and make sure there is commonality between services.
- Testing different services in a consistent manner is difficult, and the best way to work with many technologies at once is to break them down individually

XII. Team Profile

My name is Viktorija Mikelevicius. I am from Denver, Colorado, and I am in my junior year at Mines as a Computer Science major on the general track. Some of the things I am involved with include a leadership role in the Navigators ministry on campus, working in the Physical Metallurgy Lab in Hill Hall, and being a member of SWE. This past summer I had an internship at ICR analyzing Twitter data and working with a database I set up. My role in the team has been as the client liaison and the main member to update the report.

I am Camden Fritz and I am from Amherst, New Hampshire! I am a senior majoring in Computer Science at Mines and graduate in December 2023. I am a part of the Entrepreneurship and Innovation alumni interest group and Sigma Phi Epsilon. I have a deep passion for new space and have been applying my software skills to help 3D print the first rocket at a startup called Relativity Space. My role on the team was integrating messaging services and creating the performance testing suite.

Hello, I am Massimo Giusti. I am from Seattle, Washington. I am a Sophomore in Computer Science here at Mines. I am a part of Sigma Phi Epsilon here on campus. I love using software to solve interesting problems and have found a passion for software engineering ever since my internship this past summer at Microsoft. There I worked in the Developer Division, specifically on Visual Studio. I was a part of the profiler team, working on auto-analysis.

References

"Flexible & Powerful Open Source Multi-Protocol Messaging," *ActiveMQ*. [Online]. Available: https://activemq.apache.org/.

P. Luiz, "Kafka with Java, spring, and Docker - Asynchronous Communication Between Microservices," *Better Programming*, 03-Apr-2022. [Online]. Available: https://betterprogramming.pub/kafka-with-java-spring-and-docker-asynchronous-communication-between-microservices-e1d00e120831.

"Quorum queues," *RabbitMQ*. [Online]. Available: https://www.rabbitmq.com/.

"RabbitMQ Tutorials," *RabbitMQ*. [Online]. Available: https://www.rabbitmq.com/getstarted.html.

"Scaling horizontally: Kafka message bus," *IBM*, 04-Mar-2021. [Online]. Available: https://www.ibm.com/docs/en/nasm/1.1.6?topic=horizontally-kafka.

U. Khan, "Configuring Kafka SSL Using Spring Boot," *Baeldung*, 10-Oct-2022. [Online]. Available: https://www.baeldung.com/spring-boot-kafka-ssl.

Appendix A – Key Terms

Descriptions of technical terms, abbreviations, and acronyms

Term	Definition
Cloud Agnostic	*Service with cross-platform functionality across different clouds*
Microservice	*An application that either sends or receives messages, in our case*

Broker	*Facilitator of message sending between microservices. Brokers have defined queues/topics that producers can send to, that then get sent to the corresponding consumer subscribed to the topic*
Zookeeper	*Zookeeper is a process that runs alongside Kafka. Its main purpose is to manage a Kafka cluster.*
Json	*Short for JavaScript object notation*
Docker	*A service that allows for applications to be loaded onto the cloud and ran on a virtual machine.*
Virtual Machine	*A computer located in a data center (the cloud).*
Container	*A portion of a virtual machine's computing power and resources are allocated into a 'Container' that can be used to run applications.*